Hasso-Plattner-Institut für Softwaresystemtechnik
an der Universität Potsdam

Ein Beitrag zur Problematik der Integration virtueller Maschinen

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
in dem Fachgebiet Software-Engineering

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Rudolf Kugel

Potsdam, Oktober 2005

Erster Gutachter (Betreuer):	Prof. Dr.-Ing. Siegfried Wendt
Zweiter Gutachter:	Prof. Dr. rer. nat. Jürgen Nehmer
Dritter Gutachter:	Prof. Dr.-Ing. Wolfgang Effelsberg

Zusammenfassung

Heutige Softwaresysteme erlangen ihren vollen Nutzen häufig erst dann, wenn sie exakt an ihren späteren Verwendungszweck angepasst wurden. Diese Anpassung kann aufgrund der Komplexität heutiger Systeme vielfach nur noch durch Programmierung erreicht werden. Gerade in diesem Bereich bietet der Einsatz virtueller Maschinen viele Vorteile. Einerseits kann dadurch der Aufwand für die Entwicklung solcher Programme gesenkt werden, andererseits wird damit die Abhängigkeit von einer speziellen Hardwareplattform vermieden.
Obwohl die Integration virtueller Maschinen in bestehende Systeme eine große Praxisrelevanz aufweist, wird dieses Thema in der Literatur nur unzulänglich behandelt. Dies ist besonders nachteilig, weil viele im Zusammenhang mit der Integration zu treffenden Entscheidungen „trade off“ Entscheidungen sind. Über diese kann nur dann sinnvoll entschieden werden, wenn man einen Überlick über das Spektrum möglicher Lösungen hat.

Diese Arbeit verfolgt daher das Ziel, solche Zusammenhänge aufzuzeigen, die zur Beurteilung der bei einer Integration zu fällenden „trade-off“ Entscheidungen von Bedeutung sind. Die Mitarbeit an einem Forschungsprojekt innerhalb der SAP AG ermöglichte es dem Autor, die Problematik der Integration virtueller Maschinen an einem realen Praxisbeispiel kennenzulernen. Neben diesen gewonnenen Erfahrungen diente die Literatur als Informationsquelle für weitere Integrationsbeispiele sowie zur Auseinandersetzung mit der Thematik der Konstruktion virtueller Maschinen. Die Darstellung der für die Integration relevanten Zusammenhänge geschieht unter Verwendung der im Umfeld des Autors etablierten Fundamental Modeling Concepts (FMC). Der Stand der Technik auf dem Gebiet der Konstruktion virtueller Maschinen wird anhand der JAVA VM vorgestellt. Hierzu wird einerseits das in der Spezifikation definierte Funktionsprinzip der JAVA VM modelliert. Andererseits wird auf typische Konstruktionsmerkmale leistungsfähiger JAVA VM Implementierungen eingegangen. Da es aufgrund der großen Vielfalt unterschiedlicher Integrationsszenarien nicht möglich schien, alle dabei auftretenden Probleme in ein generelles Referenzmodell einzuordnen, beginnt die Diskussion der eigentlichen Integrationsproblematik mit der Darstellung der im Rahmen des Forschungsprojektes durchgeführten Integration einer JAVA VM in den SAP R/3 Application Server. Im Anschluss an dieses sehr umfangreiche Beispiel wird kurz auf weitere in der Literatur beschriebene Integrationsprojekte eingegangen. Dabei zeigt sich, dass die anhand des SAP Beispiels diskutierten Probleme keineswegs einzelfallspezifisch sind, sondern auch in anderem Kontext immer wieder auftreten. Des Weiteren wird deutlich, dass die in bestehende Systeme integrierten virtuellen Maschinen häufig dazu benutzt werden, zeitgleich mehrere „Programme“ auszuführen und dass viele aktuelle virtuelle Maschinen sich nur bedingt für einen solchen „Mehrprogrammbetrieb“ eignen, weil dies beim Entwurf der entsprechenden Maschinen nicht berücksichtigt wurde. Eine wichtige Erkenntnis dieser Arbeit ist, dass diese mangelhafte Unterstützung für einen solchen „Mehrprogrammbetrieb“ oft eines der Hauptprobleme bei der Integration darstellt. Dementsprechend wird dieses Problem im weiteren Verlauf der Arbeit besonders intensiv untersucht.

Die Leistung dieser Arbeit besteht in erster Linie darin, die Vielzahl an Problemstellungen, die bei der Integration von virtuellen Maschinen in bestehende Systeme auftreten können, so in einen Gesamtkontext einzuordnen, dass die aus diesen Problemen resultierenden Konsequenzen deutlich werden. Diese Einordnung der verschiedenen Problemstellungen richtet sich dabei speziell an Personen, die vor der Aufgabe der Integration einer virtuellen in ein bestehendes System stehen und soll ihnen den für ihre Entscheidungen notwendigen Überblick vermitteln.

Inhaltsverzeichnis

1 Einleitung

1.1 Charakterisierung des Problembereichs

Moderne Softwaresysteme sind komplexe Gebilde, welche häufig im Verbund mit anderen technischen und betriebswirtschaftlichen Systemen eingesetzt werden. Daher sind heutige Softwaresysteme oft modular aufgebaut und können vielfach mit Komponenten von unterschiedlichen Herstellern erweitert werden, um sie an den jeweiligen Verwendungszweck anzupassen. In der Vergangenheit konnte der Bedarf an Anpassungen im Vorfeld bereits eingegrenzt werden. Dementsprechend waren die für solche Erweiterungen angebotenen Mechanismen oft systemspezifisch und speziell auf diesen Bedarf ausgerichtet. Aufgrund der hohen Komplexität heutiger Systeme und der zunehmenden Vernetzung der Systeme wird es jedoch immer schwieriger, den Erweiterungsbedarf im Vorfeld der Systementwicklung abzuschätzen. Daher wird heute oft eine möglichst „universelle Anpassbarkeit“ angestrebt, was sich darin äußert, dass beim Entwurf solcher Systeme die Möglichkeit vorgesehen wird, die Systemfunktionalität später durch „Programmierung“ erweitern zu können.

Nun ist die Idee, eine Programmierschnittstelle zum Zwecke der Systemerweiterung bereitzustellen, nicht neu. Die in der Vergangenheit angebotenen Sprachen zur Formulierung dieser Erweiterungen waren oft systemspezifisch und in ihren Ausdrucksmöglichkeiten sehr beschränkt. Die Ausführung der darin formulierten Programme wurde üblicherweise mit Hilfe eines Interpreters bewerkstelligt, bei dessen Implementierung weitestgehend auf eine Übersetzung der betreffenden Programme verzichtet wurde. Aufgrund der zunehmenden Vernetzung der Systeme, des erheblichen Anstiegs der Komplexität der Systemerweiterungen und der Tatsache, dass häufig Systemerweiterungen unterschiedlicher Hersteller kombiniert werden sollen, werden heute andere Anforderungen an die zum Zwecke der Systemerweiterung angebotenen Programmiersprachen sowie an die Effizienz der Ausführung der darin formulierten Programmteile gestellt.

Um diesen Anforderungen gerecht zu werden, lassen es viele moderne Softwaresysteme zu, dass Systemerweiterungen in Standard Programmiersprachen formuliert werden können. Hierbei stellt sich jedoch die Frage, wie die entsprechenden Programmteile ausgeführt werden sollen. Eine Möglichkeit besteht darin, die betreffenden Programmteile direkt in Maschinencode zu übersetzen und diesen Code beim Start des Basissystems, beispielsweise in Form einer „dynamisch gebundenen Bibliothek“, einzubinden und unmittelbar auszuführen. Obwohl diese Vorgehensweise eine sehr effiziente Ausführung der zur Erweiterung dienenden Programmteile ermöglicht, weist sie auch gravierende Nachteile auf: Wenn der den Erweiterungen zuzurechnende Maschinencode fehlerhaft ist, kann auch die Funktionstüchtigkeit des Basissystems nicht länger garantiert werden. Darüber hinaus ist es schwierig oder gar unmöglich, aufgetretene Fehler auf die den Fehler verursachende Erweiterung zurückzuführen. Wenn das erweiterbare System auf unterschiedlichen Hardwareplattformen verfügbar sein soll, müssen die Hersteller von Systemerweiterungen ebenfalls entsprechend viele Hardwareplattformen unterstützen.

Diese Probleme lassen sich vermeiden, wenn die den Erweiterungen zuzurechnenden Programmteile mit Hilfe einer entsprechenden in das betreffende Basissystem integrierten virtuellen Maschine ausgeführt werden. Die Integration einer leistungsfähigen virtuellen Maschine in ein bestehendes Basissystem stellt jedoch selbst wieder eine anspruchsvolle Aufgabe dar. Dieser hohe Anspruch ist durch unterschiedliche Faktoren begründet: Einerseits sind die in diesem Zusammenhang zu treffenden Entscheidungen typischerweise sehr grundlegend, sodass sie spä-

ter nur unter extrem hohem Aufwand revidiert werden können. Andererseits handelt es sich bei diesen Entscheidungen oft um „trade-off“ Entscheidungen[1], deren Beurteilung sowohl detailliertes Wissen über das betreffende Basissystem als auch über den Stand der Technik auf dem Gebiet der Konstruktion virtueller Maschinen erfordert.

Den Ausgangspunkt für die Auseinandersetzung mit dem Thema der Integration virtueller Maschinen in bestehende Systeme bildete die Mitwirkung an einem industriellen Forschungsprojekt innerhalb der SAP AG, welches die Integration einer Java VM in den bestehenden SAP R/3 Application Server erforderlich machte. Die Recherche in der mit diesem Thema in Zusammenhang stehenden wissenschaftlichen Literatur führte dabei zu folgenden Ergebnissen: In der Literatur sind sehr viele virtuelle Maschinen beschrieben, die zum Zwecke der „Hardware-Plattform“-übergreifenden Verwendbarkeit von Programmen entwickelt wurden. Die Java-VM ist dabei der Kategorie der so genannten „High Level Language VMs“ zuzurechnen. Ferner ist festzustellen, dass sehr viele Beiträge existieren, die sich mit der Konstruktion solcher „High Level Language VMs“ befassen. Bei genauerer Studie dieser Beiträge stellte sich jedoch heraus, dass die Frage der Integration dieser Maschinen in bestehende Systeme entweder gar nicht oder nur am Rande behandelt wird.

Die Unzufriedenheit über diese unzureichende Behandlung dieses sehr praxisrelevanten Themas und der Umstand, dass die im Rahmen des genannten Forschungsprojektes gewonnenen Erkenntnisse zweifelsfrei auch auf andere Integrationsprojekte übertragbar sind, bildeten die Hauptmotivation für das Entstehen dieser Arbeit.

Bedenkt man die Fülle an virtuellen Maschinen und möglichen Integrationsszenarien, so ist offensichtlich, dass dieses Thema im Rahmen dieser Arbeit nicht abschließend behandelt werden kann. Vielmehr ist es das Ziel, solche Zusammenhänge aufzuzeigen, die zur Beurteilung der bei einer solchen Integration zu fällenden „trade-off“ Entscheidungen von Bedeutung sind. Dies spiegelt sich auch im Aufbau dieser Arbeit wider.

Die Dissertation gliedert sich in drei Hauptkapitel, die chronologisch aufeinander aufbauen und unterschiedliche Aspekte „virtueller Maschinen“ beleuchten.
Zunächst wird näher auf das Bedeutungsspektrum des Begriffs „virtuelle Maschine“ eingegangen. Dabei stehen speziell solche Maschinen im Vordergrund, die im Hinblick auf die plattformübergreifende Nutzung von Programmen entwickelt wurden.
Danach wird das bereits angesprochene Beispiel der Integration einer Java VM in den bestehenden SAP R/3 Application Server detailliert vorgestellt. Es handelt sich dabei um ein sehr kompliziertes Integrationsbeispiel, anhand dessen bereits eine große Vielfalt an typischen Problemstellungen behandelt werden kann.
Im Anschluss folgt die Vorstellung weiterer in der Literatur beschriebener Beispiele. Dabei zeigt sich, dass die anhand des SAP Beispiels diskutierten Probleme keineswegs einzelfallspezifisch sind, sondern auch in anderem Kontext immer wieder auftreten. Des Weiteren wird deutlich, dass die in bestehende Systeme integrierten virtuellen Maschinen häufig dazu benutzt werden, zeitgleich mehrere „Programme“ auszuführen und dass viele aktuelle virtuelle Maschinen sich nur bedingt für einen solchen „Mehrprogrammbetrieb“ eignen, weil dies beim Entwurf der entsprechenden Maschinen nicht berücksichtigt wurde. Dementsprechend wird dieses Problem im weiteren Verlauf der Arbeit besonders intensiv untersucht.

1. Entscheidungen zur Gestaltung von Systemen, bei denen man mehrere Leistungs-/Qualitätskriterien erfüllen möchte, was aber aus grundsätzlichen Strukturbeschränkungen unmöglich ist. So dass man sich entscheiden muss, in welche Richtung man tendiert.

1.2 Gliederung der Arbeit

In Kapitel 2 werden Grundlagen vorgestellt, die für das Verständnis der nachfolgenden Kapitel wichtig sind. Es besteht im Wesentlichen aus zwei Teilen.
Der Abschnitt 2.1 dient der Vorstellung des Begriffs „virtuelle Maschine“. Er wurde in den frühen 60er Jahren geprägt und bezeichnete zunächst eine spezielle Form der Simulation realer Rechnerhardware. In der weiteren Folge wurde der Begriff auch in anderem Kontext verwendet und ist heute mit unterschiedlichen Bedeutungen belegt. Die zum Zwecke von Systemerweiterungen in moderne Softwaresysteme integrierten virtuellen Maschinen gehören zur Klasse der „High Level Language VMs“.
Im Abschnitt 2.2 wird am Beispiel der „Java Virtual Machine“ genauer auf typische Konstruktionsmerkmale moderner „High Level Language VMs“ eingegangen. Hierzu wird zunächst der Inhalt der Spezifikation der Java VM vorgestellt. Darauf folgt die Vorstellung einer recht einfachen Implementierung der Firma SUN. Im Anschluss wird der Stand der Technik im Bereich der Realisierung von High Level Language VMs vorgestellt. Dabei wird ausführlich auf den Einsatz von Compiler-Technik, sowie auf die Abbildung von Java Threads auf Betriebssystemthreads eingegangen. Ferner wird das Thema „Garbage Collection“ angesprochen.
Dass die Java VM als Beispiel ausgewählt wurde, hat verschiedene Gründe: Einerseits ist die Konstruktion der Java VM recht einfach und weist dennoch die typischen Merkmale moderner High Level Language VMs auf. Andererseits ist die Java VM die wohl am ausführlichsten untersuchte High Level Language VM, die in der Literatur zu finden ist. Darüber hinaus wäre es im Hinblick auf die in Kapitel 3 beschriebene Integration der Java VM in den SAP Application Server ohnehin erforderlich gewesen, detaillierter auf die Eigenschaften der Java VM einzugehen.

In Kapitel 3 wird das Thema der Integration virtueller Maschinen anhand des bereits erwähnten Beispiels der Integration der Java/J2EE Technologie in den SAP Application Server vorgestellt. Wie bereits eingangs erwähnt, unterliegt die Art der Integration virtueller Maschinen in bestehende Systeme oft vielen „trade off“ Entscheidungen. Um die im Fall dieser Integration getroffenen Entscheidungen verstehen zu können, wird zu Beginn von Kapitel 3 zunächst genauer auf die Hintergründe der Integration eingegangen. In diesem Zusammenhang wird ein Vergleich der bestehenden SAP Application Server Technologie mit der Java basierten J2EE Application Server Technologie angestellt. Außerdem werden die mit der Integration verfolgten Ziele kurz dargelegt. Im Anschluss folgt die Darstellung der technischen Umsetzung der Integration. Aufgrund der Tatsache, dass die Java VM bereits in Kapitel 2 behandelt wurde, ist die Darstellung der Integrationsproblematik sehr kompakt und setzt sich gezielt mit den bei der Integration zu lösenden Problemen auseinander.

In Kapitel 4 werden die Erkenntnisse aus dem SAP Projekt mit denen aus anderen in der Literatur beschriebenen Integrationsvorhaben verglichen.
Im ersten Teil von Kapitel 4 werden die zum Vergleich herangezogenen Integrationsbeispiele grob umrissen. Dabei wird die Integration von virtuellen Maschinen in Web-Browser, relationale Datenbanksysteme und Agentensysteme angesprochen. Ferner werden noch zwei weitere Systeme zur Sprache gebracht, die mit dem SAP Beispiel in Zusammenhang stehen. Dabei handelt es sich zum einen um den von der Firma Gemstome entwickelten Application Server mit der Produktbezeichnung „GemStone Facets“, zum anderen wird die „PEVM“ vorgestellt. Die „PEVM“ ist eine spezielle Java VM, die über einen „Checkpoint“-Mechanismus verfügt.
Im zweiten Teil von Kapitel 4 werden die bei den verschiedenen Integrationsbeispielen angesprochenen Probleme verglichen und mögliche Lösungen für typische Probleme diskutiert. Wie bereits erwähnt, werden die in bestehende Systeme integrierten virtuellen Maschinen häufig

dazu benutzt, zeitgleich mehrere „Programme“ auszuführen. Ebenso wie viele andere „High Level Language VMs“ ist die Java VM hierfür jedoch nur bedingt geeignet. Um diesen „Mehrprogrammbetrieb“ in der gewünschten Form durchführen zu können, ist es daher meist unumgänglich, die Integration so zu gestalten, dass mehrere VMs parallel betrieben werden können. Dabei stellt es eine große Herausforderung dar, den für den Betrieb dieser vielen VMs notwendigen Betriebsmittelbedarf auf ein erträgliches Maß zu reduzieren. Aufgrund der großen praktischen Relevanz dieses Problems konzentriert sich die zweite Hälfte von Kapitel 4 fast ausschließlich auf die Darstellung der in der Literatur beschriebenen Möglichkeiten zur Reduktion dieses Betriebsmittelbedarfs. Dabei lassen sich grundsätzlich verschiedene Herangehensweisen unterscheiden. Einerseits wird darauf eingegangen, wie der Speicherbedarf durch gezieltes „sharing“ von solchen für alle VMs gleichermaßen relevanten Daten reduziert werden kann. Eine zweite Herangehensweise, die vor allem dann zum Einsatz kommt, wenn die Dauer der Ausführung eines Programms kurz ist, besteht darin, eine einmal erzeugte VM nacheinander zur Ausführung verschiedener Programme zu verwenden. Wieder andere Ansätze verfolgen das Ziel, solche für den Mehrprogrammbetrieb besser geeignete VMs zu konstruieren, indem sie die Konzepte von „High Level Language VMs“ mit dem aus dem Bereich der Betriebssysteme bekannten Prozesskonzept kombinieren.

2 Virtuelle Maschinen

2.1 Historische Entwicklung des Begriffs Virtuelle Maschine

Der Begriff virtuelle Maschine kam in den frühen 60er Jahren auf. Speziell die IBM hat mit ihrer Rechnerfamilie IBM /360 und der dazu angebotenen Software wesentlich zur Prägung und Verbreitung des Begriffs beigetragen. Die Motivation für die Einführung des Konzeptes *virtueller Maschinen* bestand darin, dass man mit den Rechnern der IBM 360 Familie einerseits ein breites Preis-/Leistungsspektrum abdecken wollte und andererseits die Austauschbarkeit von Programmen zwischen den verschiedenen Rechnermodellen gewährleisten musste. Das Ziel, die Austauschbarkeit von Programmen zwischen den verschiedenen Rechnermodellen der IBM /360 Familie zu gewährleisten, wurde einerseits dadurch verfolgt, dass bei der Konstruktion der Hardware entsprechende Kompatibilität „hineinkonstruiert" wurde. Diese konstruktiven Maßnahmen umfassten insbesondere die Festlegung eines Standard-Befehlssatzes für die Zentralprozessoren der Rechner der IBM /360 Familie. Das Ziel der Softwarekompatibilität wurde, neben den die Hardware betreffenden Maßnahmen, speziell auch dadurch verfolgt, dass man eine „Virtual Machine Monitor"-Software entwickelte, die es erlaubte, auf einem realen Rechner der IBM /360 Familie zeitgleich mehrere Rechner der IBM /360 Familie zu simulieren. Diese Software hat ihren Ursprung im Systemkern CP 67 genommen. Die Nachfolger des Systemkerns CP 67 waren später unter dem Namen „VM" bekannt.
Heute wird der Begriff virtuelle Maschine vielfach und in unterschiedlichen Bedeutungen verwendet. So wird er beispielsweise auch im Maschinen- und Anlagenbaubereich verwendet und bezeichnet dort Systeme, die der Simulation des Verhaltens von technischen Systemen wie Werkzeugmaschinen oder Produktionsanlagen dienen. Aber auch im Bereich der Informationstechnik wird der Begriff heute in wesentlich allgemeinerem Sinne verwendet als damals. Bevor der Bedeutungswandel des Begriffs im Bereich der Informationstechnik beschrieben wird, soll zunächst die Bedeutung des Begriffs *virtuell* geklärt werden. Eine ausführliche Auseinandersetzung mit dem Begriff virtuell findet man in [Scholz 00]:

Der Begriff der Virtualität

> *„ ... Virtuell (abgeleitet vom lateinischen „virtus" = „Tüchtigkeit") steht dabei für nicht wirklich, scheinbar oder der Anlage nach vorhanden. Als virtuell wird die Eigenschaft einer Sache bezeichnet, die zwar nicht real, aber doch in der Möglichkeit existiert; Virtualität spezifiziert also ein konkretes Objekt über Eigenschaften, die nicht physisch, trotzdem ihrer Leistungsfähigkeit nach vorhanden sind.*
> *Virtualisierung ist ein generelles Konzept. Virtualität benötigt deshalb auch immer einen spezifizierten Bezug zu einem konkreten Objekt: Es gibt demnach keine Virtualität per se, sondern lediglich virtuelle Fernsehstudios, virtuelle Produkte, ... "*

Nach Scholz ist die Definition virtueller Objekte immer mit einem Virtualisierungsprozess verbunden, der ausgehend von einem realen Objekt die Eigenschaften des virtualisierten Objekts spezifiziert. Des Weiteren definiert Scholz vier charakteristische Merkmale, die die Beziehung zwischen dem Ausgangsobjekt und dem virtuellen Gegenstück kennzeichnen:

- *„Der Virtualisierungsprozeß beginnt immer mit der Spezifizierung eines zu virtualisierenden Objektes über seine Merkmale. Diese **konstituierenden Charakteristika** weisen sowohl das ursprüngliche (reale) Objekt als auch seine virtuelle Realisierung auf."*

- *„Anschließend läßt sich festlegen, welche Attribute virtualisiert werden: Entscheidend bei der Idee der Virtualisierung ist immer das **Fehlen von bestimmten physikalischen Attributen des ursprünglichen Objektes**, die üblicherweise mit dem zu virtualisierenden Objekt assoziiert werden, aber beim virtuellen Objekt nicht mehr vorhanden sind und trotzdem in ihrer erlebbaren Funktionalität realisiert werden.“*
- *„Dies läßt sich allerdings nur durch entsprechende **spezielle Zusatzspezifika** verwirklichen, wobei es sich oft (aber nicht immer!) um technische Hilfsmittel handelt.“*
- *„Ergebnis ist ein **Nutzenvorteil**, der sich durch den Wegfall der physikalischen Attribute ergibt.“*

Virtuelle IBM /360 Maschinen

Im Fall der durch Verwendung des Systemkerns VM geschaffenen „virtuellen IBM /360 Maschinen“ bildeten die realen IBM /360 Maschinen den Ausgangspunkt für den Virtualisierungsprozess. Die Gesamtheit der konstituierenden Charakteristika eines (virtuellen) Rechners der IBM /360 Familie war durch die informationelle Struktur gegeben, die ein Systemprogrammierer an der Hardware/Softwareschnittstelle erlebte, wenn er Software für einen realen Rechner der IBM /360 Familie entwickelte.
Der Aufwand, der zur Entwicklung von Virtual Machine Monitor Software für einen bestimmten Rechnertyp notwendig ist, hängt wesentlich von der Beschaffenheit der Hardware/Softwareschnittstelle ab. Ebenso ist die spätere Effizienz des Virtual Machine Monitors, welche sich insbesondere im Verhältnis zwischen der vom Virtual Machine Monitor verbrauchten Rechenzeit und der den virtuellen Rechnern zur Verfügung gestellten Rechenzeit äußert, von der Beschaffenheit Hardware/Softwareschnittstelle abhängig. Der Bedarf nach der Virtual Machine Monitor Software hat daher wesentlichen Einfluss auf den Befehlssatz der realen CPU und die Art und Weise der Peripherieankopplung gehabt. Außerdem spielt das Konzept des „virtuellen Speichers“, dessen Entwicklung mit der Entwicklung des Konzeptes virtueller Maschinen einherging, eine sehr wichtige Rolle bei der Realisierung von Virtual Machine Monitor Software.
Die Verfügbarkeit eines derart effizienten Virtual Machine Monitors brachte eine Reihe von Nutzenvorteilen mit sich. Der wohl wichtigste Vorteil war der, dass es möglich wurde, zeitgleich mehrere verschiedene Betriebssysteme auf einem Rechner zu betreiben und die jeweils dafür entwickelten Anwendungsprogramme zu nutzen. Auch beim Umstieg auf neue Softwareversionen - insbesondere auch auf neue Betriebssystemversionen - erwies es sich als vorteilhaft, die neue Version der Software zunächst auf einer virtuellen Maschine installieren und testen zu können. Und nicht zuletzt war auch der Umstieg auf leistungsfähigere Hardware einerseits relativ unproblematisch und andererseits wirkte sich ein solcher Umstieg gleichermaßen positiv auf das Laufzeitverhalten neuer und bereits vorhandener Software aus.

Heutige Verwendung des Begriffs virtuelle Maschine

Die Motivation für die Virtualisierung von IBM /360 Rechnern bestand darin, dass man Softwarekompatibilität zwischen den verschiedenen Modellen der IBM /360 Familie erreichen wollte. Seit den 60er Jahren hat die Komplexität von Softwaresystemen rasant zugenommen und die plattformübergreifende Verwendbarkeit von Software hat größere Bedeutung als je zuvor. Bei den klassischen virtuellen Maschinen ging es darum, auf einer realen Maschine mehrere virtuelle Maschinen zu simulieren. Die simulierten Maschinen wurden darüber hinaus nur dann als virtuelle Maschinen bezeichnet, wenn die konstituierenden Charakteristika dieser Maschinen durch die Hardware/Softwareschnittstelle der zur Simulation verwendeten realen Maschine definiert waren und eine ganz bestimmte Simulationstechnik verwendet wurde. Inzwischen wird der Begriff virtuelle Maschine auch dann verwendet, wenn die konstituieren-

den Charakteristika der simulierten Maschine aus der Hardware/Softwareschnittstelle eines beliebigen anderen Rechnertyps oder gar nicht aus einer Hardware/Softwareschnittstelle einer real existierenden Maschine abgeleitet werden. Man hat nämlich sehr bald erkannt, dass man das Ziel der plattformübergreifenden Verwendbarkeit von Software auf andere Weise viel leichter erreichen kann. Die grundsätzliche Vorgehensweise besteht dabei darin, dass man zunächst eine abstrakte Maschine definiert und diese dann als Zielplattform für die Softwareentwicklung verwendet. Weil man bei der Definition der konstituierenden Charakteristika einer solchen abstrakten Maschine nicht an Eigenschaften einer bestimmte Hardware gebunden ist, kann man sie so wählen, dass die Simulation einer solchen Maschine auf allen in Frage kommenden Hardwareplattformen leicht möglich ist. Historisch gesehen stand die Entwicklung solcher Maschinen fast immer im Zusammenhang mit der Entwicklung von höheren Programmiersprachen. Daher werden sie in der Literatur auch häufig als „High Level Language VM's" bezeichnet.

In dem nun folgenden Abschnitt 2.1.1 werden die Konzepte, die bei der Realisierung klassischer virtueller Maschinen Verwendung fanden, unter Bezug auf die Rechner der IBM /360 Familie detaillierter vorgestellt. Die Wahl der IBM /360 Familie als Bezugspunkt für die Darstellung ist einerseits dadurch begründet, dass die IBM damals technologisch führend war, andererseits wird in vielen Veröffentlichungen zum Thema virtuelle Maschinen ebenfalls auf die IBM /360 Familie Bezug genommen. Im Anschluss an die Vorstellung der klassischen Bedeutung des Begriffs wird in Abschnitt 2.1.2 das heutige Spektrum der Virtualisierung von Maschinen vorgestellt, deren konstituierende Charakteristika einen direkten Bezug zu realer Hardware haben. In Abschnitt 2.1.3 wird schließlich auf das Design von „High Level Language VM's" eingegangen.

2.1.1 Klassische virtuelle Maschinen

Die Forschung, die auf dem Gebiet virtueller Maschinen in den 60er Jahren und zu Beginn der 70er Jahre stattfand, ist in zahlreichen Veröffentlichungen festgehalten. Der Inhalt dieses Abschnitts orientiert sich teilweise an dem Aufsatz „Survey of Virtual Machine Research" [Goldberg 74], der einen guten Überblick über die Forschungsaktivitäten dieser Zeit gibt. Bild 1 zeigt eine in diesem Aufsatz vorkommende Graphik, die die Beziehungen zwischen einer realen und verschiedenen virtuellen Maschinen veranschaulicht. Jeder der grau hinterlegten Bereiche symbolisiert eine virtuelle oder reale Maschine. Ganz oben im Bild ist die mit „R" bezeichnete reale Maschine dargestellt. Auf dieser wird ein Virtual Machine Monitor betrieben, der der Simulation der beiden darunter dargestellten virtuellen Maschinen „V_1" und „V_2" dient. Die virtuelle Maschine „V_1" wird wiederum dazu benutzt einen Virtual Machine Monitor zu betreiben, der die virtuellen Maschinen „$V_{1.1}$" und „$V_{1.2}$" simuliert. Auf den virtuellen Maschinen „$V_{1.1}$", „V_2" und „$V_{1.2}$" sind verschiedene Betriebssysteme im Einsatz, die, je nach Betriebssystem verschieden, entweder eine oder mehrere „erweiterte Maschinen" zum Betrieb von Benutzerprogrammen bereitstellen können. Die erweiterten Maschinen weisen im Unterschied zu den virtuellen Maschinen eine von der realen Maschine abweichende, für das jeweilige Betriebssystem spezifische Schnittstelle auf.

Der Begriff der „virtuellen Maschine" wurde streng von einer irgendwie gearteten Simulation einer Maschine unterschieden. Popek und Goldberg haben das damalige Verständnis des Begriffs in folgender Definition zusammengefasst:

> *„A virtual machine is taken to be an efficient, isolated duplicate of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM). As a piece of*

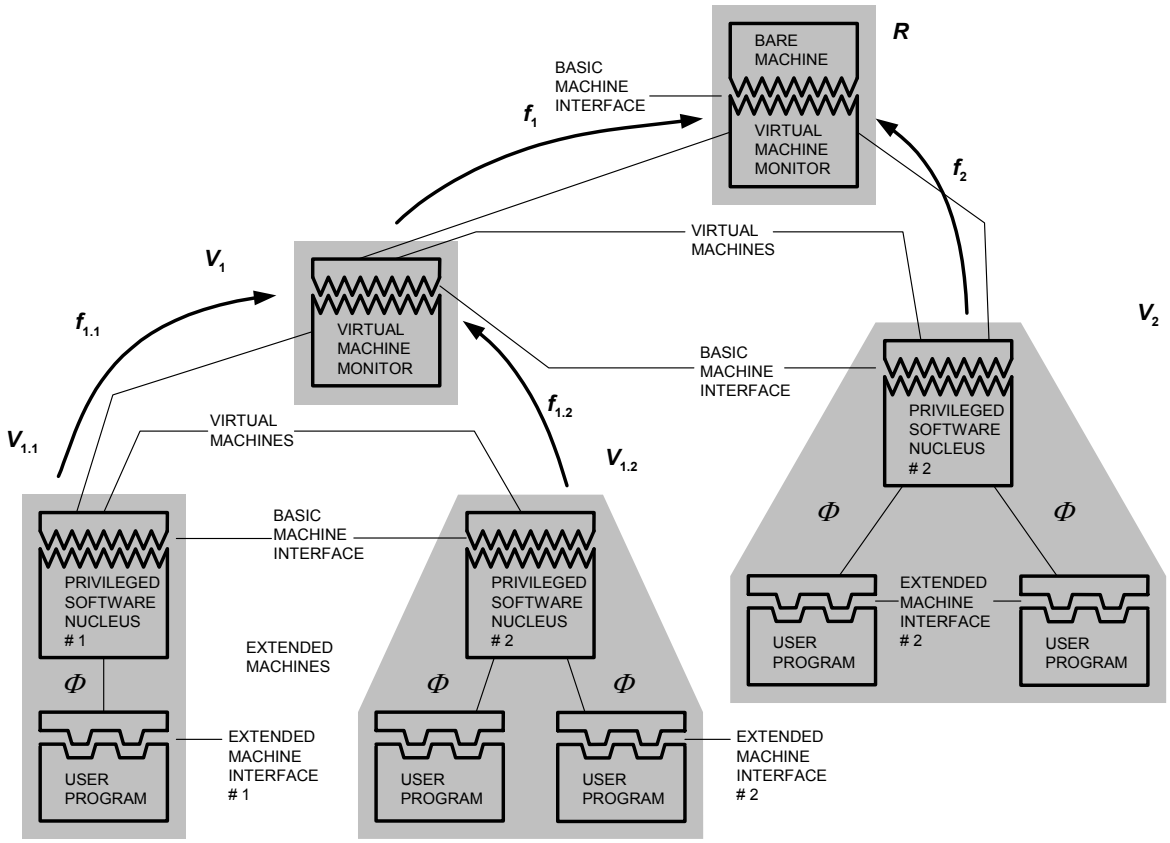


Bild 1: „Virtual Machine Model with Recursion“ nach [Goldberg 74]

> *software a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last the VMM is in complete control of system resources.*
> *By an „essentially identical“ environment, the first characteristic, is meant the following. Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system ressources and by timing dependencies.*
> *...*
> *The second characteristic of a virtual machine monitor is efficiency. It demands that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor without software intervention by the VMM.*
>
> *...*
> *A virtual machine is the environment created by the VMM. ...“* [Popek Goldberg 74]

Die erste Forderung der Definition legt fest, dass es sich bei einer virtuellen Maschine stets um ein per Simulation geschaffenes „Duplikat“ derjenigen realen Maschine handelt, die zur Simulation benutzt wird. Die virtuelle Maschine und die zur Simulation benutzte Maschine dürfen sich an ihrer für einen Systemprogrammierer relevanten Schnittstelle nur durch die Menge an verfügbaren Systemressourcen unterscheiden, nicht jedoch in den übrigen Eigenschaften. Unter Systemressourcen sind in diesem Zusammenhang Hauptspeicher, Prozessorzeit und insbesondere auch Peripheriegeräte wie Festplatten, Bandlaufwerke, Drucker etc. zu verstehen.
Aus der zweiten Forderung nach der Simulationsgeschwindigkeit ergeben sich letztlich Anforderungen an die Rechnerhardware. Erfüllt eine Rechnerarchitektur diese Anforderungen, so

wird sie auch als „Virtualizable Architecture“ bezeichnet. Die Anforderungen an eine „Virtualizable Architecture“ sowie die für VMM-Software kennzeichnenden Implementierungskonzepte werden in den Abschnitten 2.1.1.3 und 2.1.1.2 vorgestellt. Bevor auf die besonderen Anforderungen eingegangen werden kann, die über die Virtualisierbarkeit einer Rechnerarchitektur entscheiden, werden im Abschnitt 2.1.1.1 zunächst die generellen Eigenschaften der damaligen Rechnerhardware vorgestellt sowie einige spezielle Eigenschaften der IBM /360 Hardware.
Die dritte Anforderung an die VMM Software ist eine Anforderung, die in ähnlicher Weise auch an Betriebssysteme gestellt wird. Sie besteht darin, die Hardware-Ressourcen der realen Maschine zu verwalten und dabei die gegenseitige Abschottung der virtuellen Maschinen zu gewährleisten. Im Unterschied zu Betriebssystemen ist die Abbildung zwischen den, den virtuellen Maschinen zur Verfügung gestellten, „virtuellen Ressourcen“ und den realen Ressourcen im Allgemeinen einfacher. Ein Grund dafür ist, dass man die VMM Software möglichst einfach halten wollte, um die Fehlerquellen in der VMM Software zu reduzieren und die Intervention des VM-Monitors bei I/O-Vorgängen gering zu halten. Daher war für einige der Peripheriegeräte nur eine exklusive Zuordnung zu einer der virtuellen Maschinen möglich. Ein weiterer Grund für die eingeschränkte Unterstützung des Multiplexbetriebs von Geräten durch die VMM-Software ist der, dass die an der Hardware/Softwareschnittstelle einer Maschine zur Verfügung stehende Information dafür im Allgemeinen nicht ausreichend ist. Für die IBM /360 Systeme war jedoch, neben der Unterstützung für die Multiplexnutzung von Hauptspeicher und Prozessor, auch die Multiplexnutzung von Festplatten durch verschiedene virtuelle Maschinen selbstverständlich.
Die Abbildung der Ressourcen der virtuellen Maschinen „$V_{1.1}$“ und „$V_{1.2}$“ auf die Ressourcen der realen Maschine „R“ aus Bild 1 geschieht zweistufig. Zuerst müssen diese virtuellen Ressourcen auf Ressourcen der Maschine „V_1“ abgebildet werden, bevor die Abbildung auf die physischen Ressourcen erfolgen kann. Die Unterstützung dieser „rekursiven“ Verwendbarkeit der Virtual Machine Monitor Software war ein in dieser Zeit viel diskutiertes Thema und hat zu verschiedenen Vorschlägen für die Hardwareunterstützung solcher mehrstufigen Abbildungen geführt.[2]

2.1.1.1 „Third Generation Architecture“

Obwohl die Eigenschaften der damaligen Rechnerhardware stark vom Hersteller abhängig war, wiesen die verschiedenen Modelle doch wesentliche Gemeinsamkeiten auf. Diese Eigenschaften wurden unter dem Begriff „Third Generation Architecture“ zusammengefasst:

(1) Das System beinhaltet genau einen multiplexfähigen, sequentiell arbeitenden Hauptprozessor (CPU).

(2) Alle Third Generation Architectures unterstützen den Einsatz von Betriebssystemen zur Betriebsmittelverwaltung. Dazu wurden die Prozessoren dieser Systeme so konstruiert, dass sie über zwei Betriebsarten sowie über entsprechende Vorkehrungen zum Wechsel zwischen diesen beiden Betriebsarten verfügten. Wenn aktuell die Betriebsart „privileged mode“ eingestellt ist, ist der Zugriff auf alle im Rechner vorhandenen Ressourcen uneingeschränkt möglich. Befindet sich der Prozessor hingegen in der Betriebsart „non privile-

2. Eine mehrstufige Virtualisierung des Prozessors durch VMM Software ist ohne spezielle Hardwareunterstützung praktisch nicht möglich, da der Virtualisierungsaufwand meist nicht linear mit der Hierarchietiefe, sondern stärker wächst. Ein weiteres Problem bei der rekursiven Virtualisierung besteht darin, dass die VMM’s auf verschiedenen Hierarchieebenen nur dann eine sinnvolle Verwaltung von I/O Puffern betreiben können, wenn sie ihre Position innerhalb der Virtualisierungshierarchie kennen.

ged mode“, verhindert er die Veränderung von bestimmten Einstellungen, die mit der Betriebsmittelverwaltung in Zusammenhang stehen. Um die betreffenden Einstellungen vor Änderungen im „non privileged mode“ schützen zu können, sind die Prozessoren so konstruiert, dass diese Einstellungen nur durch Ausführung eigens dafür vorgesehener Maschinenbefehle verändert werden können. Solche Befehle werden auch als privilegierte Befehle bezeichnet und führen nur bei Anwendung in der Betriebsart „privileged mode“ zu Veränderungen an den geschützten Einstellungen. Der Versuch, einen privilegierten Befehl im „non privileged mode“ anzuwenden, führt je nach Prozessortyp zu unterschiedlichen Konsequenzen. Manche Prozessoren ignorieren den privilegierten Befehl einfach, bei anderen Prozessoren hat der Befehlscode des privilegierten Befehls im „non privileged mode“ eine andere Bedeutung. Fortschrittliche Prozessoren bringen in diesem Fall eine vorher zu definierende Ausnahmebehandlungsroutine zur Ausführung und schalten automatisch in den „privileged mode“ um. Sämtliche im „privileged mode“ ausgeführte Software ist solche, die zum Betriebssystem gehört und wird im Aufsatz von Goldberg auch als „privileged Software nucleus“ bezeichnet. Normale Benutzerprogramme werden grundsätzlich im „non privileged mode“ ausgeführt. Da alle zur Betriebsmittelverwaltung notwendigen Befehle als privilegierte Befehle realisiert sind, erhält das Betriebssystem die Kontrolle über sämtliche Betriebsmittel. Zur Anforderung von Betriebsmitteln durch Benutzerprogramme stellen die Prozessoren einen speziellen „supervisor call (SVC)“ Maschinenbefehl zur Verfügung. Die Ausführung dieses Befehls führt zur definierten Umschaltung auf eine vom Betriebssystem bereitzustellende Routine und zur Umschaltung in den „privileged mode“.

(3) Einheitlicher, linear adressierbarer Hauptspeicher
Einheitlich heißt hier: Keine Trennung zwischen Programm und Datenspeicher.

(4) Hardwareunterstützung für virtuelle Adressierung
Zu dieser Zeit waren zwei verschiedene Arten der Unterstützung verbreitet:

a) Segmentbasierter virtueller Adressraum
Diese Art der virtuellen Adressierung wird durch zwei Prozessorregister unterstützt: Basis- und Grenzregister („relocation / bounds register“). Die reale Adresse wird durch Addition der virtuellen Adresse und dem Inhalt des Basisregisters errechnet. Ist die virtuelle Adresse größer als die durch den Inhalt des Grenzregisters festgelegte Segmentgröße liegt eine Speicherschutzverletzung vor.

b) Seitenbasierter virtueller Adressraum
Sowohl der reale Adressraum als auch die virtuellen Adressräume werden in aufeinanderfolgende, gleichgroße Seiten eingeteilt. Zu jedem virtuellen Adressraum gibt es eine Seitentabelle, die definiert, welche reale Speicherseite den Inhalt der betreffenden virtuellen Seite enthält. (Die Seitentabellen befinden sich ebenfalls im Hautspeicher.) Falls einer Seite des virtuellen Adressraums keine reale Speicherseite zugeordnet ist, liegt eine Speicherschutzverletzung vor.

Hardwarearchitektur eines IBM /360 Systems

Die an der Hardware/Softwareschnittstelle eines IBM /360[3] Rechners erlebbare informationelle Struktur wird jetzt anhand von Bild 2[4] näher vorgestellt. In der oberen Bildhälfte ist der Haupt-

3. Die im Folgenden über Rechner der IBM /360 Familie gemachten Aussagen treffen in dieser Form nur auf einen Teil der Familienmitglieder zu. In ganz frühen Familienmitgliedern war noch keine Hardwareunterstützung für seitenbasierte virtuelle Adressierungsverfahren vorhanden. Späte Rechner der IBM /360 Familie konnten bereits mit mehr als sieben Transferprozessoren sowie mit einem zweiten Prozessor ausgestattet werden.

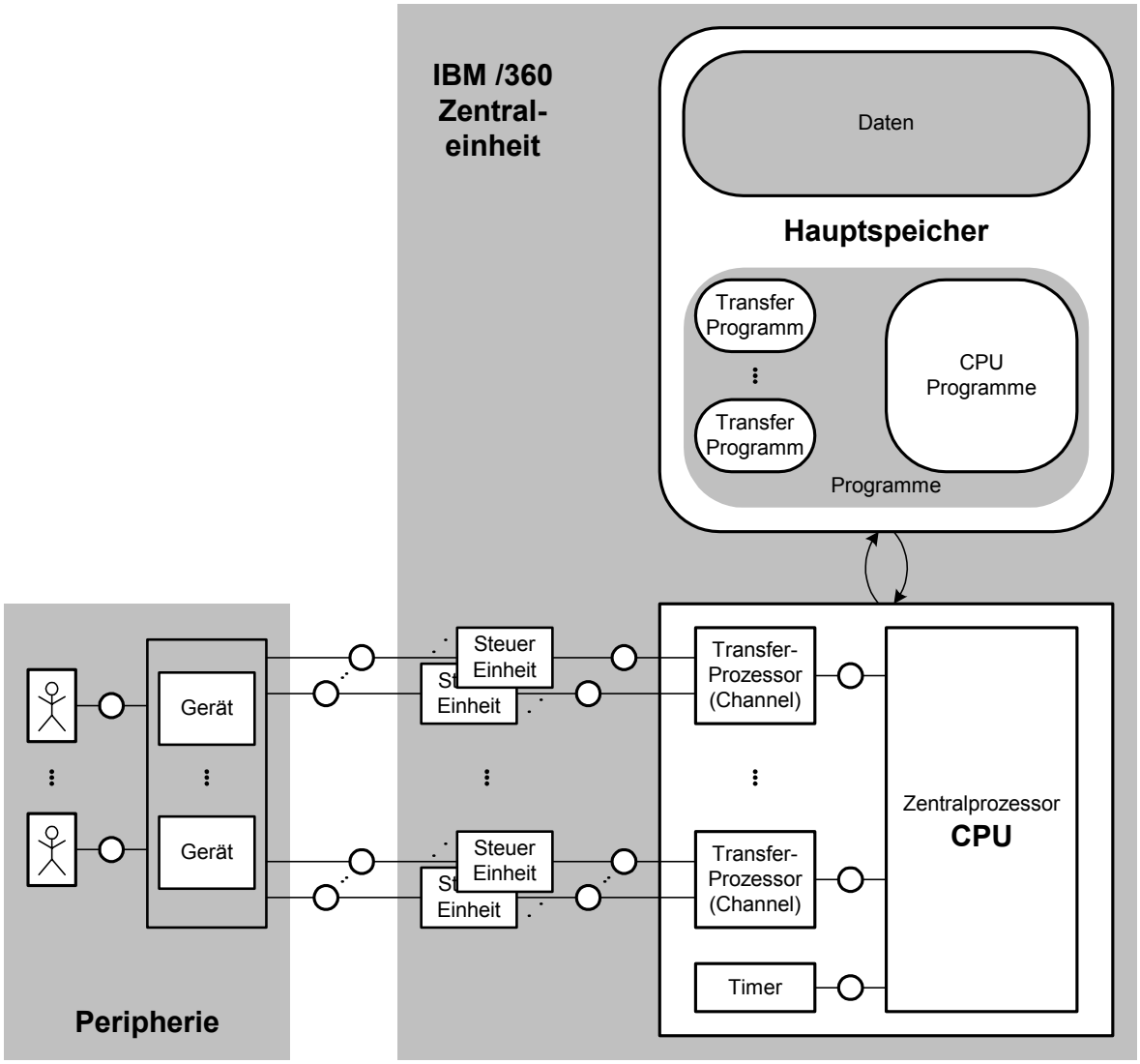


Bild 2: Prinzipieller Aufbau eines IBM /360 Systems[4]

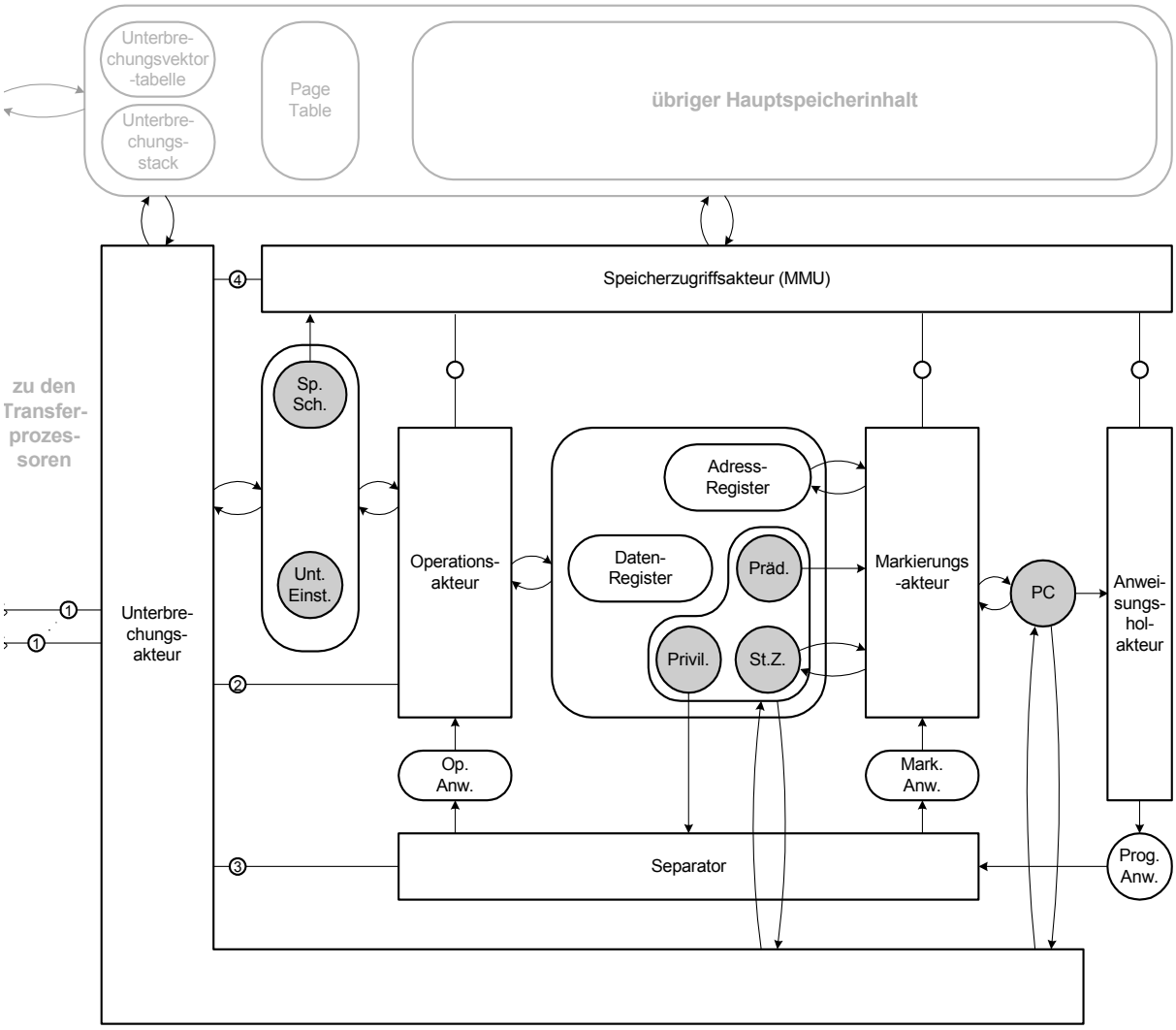
speicher dargestellt. Wie für eine „Third Generation Architecure“ typisch gibt es keine Trennung zwischen Programm und Datenspeicher. Der linear adressierbarer Hauptspeicher, der als die zentrale Systemkomponente angesehen werden kann, liegt im Zugriff mehrerer unterschiedlicher Prozessoren. Unter diesen Prozessoren gibt es einen ausgezeichneten Zentralprozessor (CPU). Er übernimmt die eigentliche Datenverarbeitung sowie die Koordination der Transferprozessoren. Während der Zentralprozessor für die Verknüpfung von Daten zuständig ist, dienen die Transferprozessoren dazu, den Transport von Daten zwischen Hauptspeicher und Peripherie abzuwickeln. Je nach Ausstattung kann ein IBM /360 System bis zu sieben Transferprozessoren beinhalten, über die jeweils bis zu 256 Geräte angebunden sind. Die Transferprozessoren eines IBM /360 Systems, welche in der Originalliteratur auch als „Channels“ bezeichnet werden, können nebenläufig zum Zentralprozessor arbeiten und sind normalerweise

4. Bei Bild 2 handelt es sich um einen FMC Aufbauplan. Das Akronym FMC steht stellvertretend für „**F**undamental **M**odeling **C**oncepts“ und bezeichnet die Modellierungstechnik, die zur Darstellung der für diese Arbeit relevanten Zusammenhänge benutzt wurde. Als Einführung in die Modellierungstechnik FMC empfiehlt sich [FMC 05], [FMC WWW]. Weitere Veröffentlichungen zu FMC: [Bungert 98], [Gröne et al 03], [Hecker 81], [Keller et al 02], [Keller, Wendt 03], [Kleis 99], [SAP 90-01], [Tabeling 02], [Wendt 79], [Wendt 91], [Zuck 90].

unterschiedlichen Typs. Ein „Multiplexor Channel“ dient zur Anbindung zeichenorientierter Geräte, während ein „Selector Channel“ oder ein „Block Multiplexor Channel“ zur Anbindung blockorientierter Geräte dient. „Multiplexor Channels“ und „Block Multiplexor Channels“ sind für den Mehrprogrammbetrieb geeignet und können daher zeitgleich mehrere Transferprogramme für unterschiedliche Geräte abarbeiten. Der Austausch von Daten mit der Peripherie findet grundsätzlich nur über den Hauptspeicher statt. Insbesondere fließen über die Kanäle zwischen den Transferprozessoren und dem Zentralprozessor keine mit der Peripherie auszutauschenden Daten sondern nur solche Daten, die zur Synchronisation der Prozessoren notwendig sind. Der grundsätzliche Ablauf beim Austausch von Daten zwischen Peripherie und Hauptspeicher ist daher der Folgende: Bevor der Datenaustausch stattfinden kann, muss das Transferprogramm für den Channel im Hauptspeicher zur Verfügung gestellt werden. Außerdem müssen die zu transportierenden Daten bzw. Pufferbereiche zur Aufnahme der zu beschaffenden Daten im Hauptspeicher zur Verfügung gestellt bzw. reserviert werden. Anschließend stößt der Zentralprozessor durch Ausführung eines „SIO“-Maschinenbefehls die asynchrone Abwicklung des Transferauftrags durch den gewünschten Channel an. Wenn der Channel einen Transferauftrag fertig abgewickelt hat, teilt er dies dem Zentralprozessor durch das Auslösen eines Interrupts mit. Eine weitere für die Organisation des Rechenbetriebs wichtige Systemkomponente ist der Timer, der ebenfalls Interrupts auslösen kann.

Modell des Systemprozessors

Die Anforderung an die Effizienz von VMM Software ist untrennbar mit einer ganz bestimmten Implementierungstechnik verbunden, deren Erklärung ein detaillierteres Modell von der Funktionsweise des Prozessors voraussetzt. Die angesprochene Implementierungstechnik ist unabhängig von der verwendeten Technik zur Virtualisierung des Adressraums, im Folgenden wird daher ein seitenbasiertes Verfahren angenommen. Bild 3 zeigt ein entsprechendes Modell eines Prozessors, welches stark an Bild 252 auf Seite 472 in [Wendt 91] angelehnt ist. Der Aufbau zeigt fünf Akteure, die zusammen die Aufgabe des Prozessors übernehmen und dabei auf verschiedenen Speichern operieren. Der im Bild ganz rechts dargestellte Anweisungsholakteur ist dafür zuständig, den nächsten auszuführenden Maschinenbefehl zu holen. Dazu liest er den aktuellen Wert des Programmzählers, der im Bild als Speicher mit der Bezeichnung „PC“ dargestellt ist, aus. Dieser Speicher enthält die virtuelle Adresse ab der der nächste zu holende Befehl im Speicher zu finden ist. Danach holt er diesen Befehl, indem er den Speicherzugriffsakteur nacheinander um die Inhalte der den Befehl beinhaltenden Speicherzellen bittet, und legt ihn im Speicher mit der Bezeichnung „Prog. Anw.“ ab. Anschließend trennt der Separator den Maschinenbefehl in eine Operations- und eine Markierungsanweisung auf. Die Operationsanweisung übergibt er zur Ausführung an den Operationsakteur, der die entsprechende Operation auf den in der Operationsanweisung angegebenen Operanden ausführt und dabei die Werte der Verzweigungsprädikate anpasst. Bei den Operanden eines Befehls kann es sich einerseits um die Inhalte der Prozessorregister handeln, die rechts und links vom Operationsakteur dargestellt sind. Andererseits kann der Operationsakteur über den Speicherzugriffsakteur auch auf Operanden zugreifen, die im Hauptspeicher liegen. Der Markierungsakteur, der vom Separator die Markierungsanweisung zur Ausführung erhält, ist für die Bestimmung der virtuellen Adresse des nächsten zu holenden Befehls zuständig. Der Markierungsakteur hat Zugriff auf die vom Operationsakteur bereitgestellten Verzweigungsprädikate sowie auf die Adressregister. Des Weiteren hat er, zur Unterstützung von Unterprogrammtechnik, Zugriff auf einen Stapelzeiger und über den Speicherzugriffsakteur auch auf den im Hauptspeicher befindlichen Stapelinhalt. Die Aufgabe des Speicherzugriffsakteurs besteht darin, die an ihn gerichteten, unter Bezug auf virtuelle Adressen formulierten Hauptspeicherzugriffe durchzuführen. Bevor er die angeforderten Zugriffe durchführen kann, muss er die in den Zugriffsaufträgen vorkommenden virtuellen



Bedeutung der Abkürzungen:

Mark.Anw.:	Markierungsanweisung	Prog.Anw.:	Programmanweisung / Maschinenbefehl
Op. Anw.:	Operationsanweisung	Sp. Sch.:	Speicherschutz Einstellungen
PC:	Programmzähler	St.Z.:	Stapelzeiger / Stackpointer
Präd.:	Verzweigungsprädikate	Unt.Einst.:	Unterbrechungseinstellungen
Priv.:	Ausführungsprivileg / Betriebsart		

Bild 3: Aufbau eines multiplexfähigen Prozessors

Adressen zunächst in reale Adressen umrechnen. Außerdem muss er die möglicherweise mit der virtuellen Speicherseite verknüpften Zugriffseinschränkungen dahingehend überprüfen, ob die angeforderte Art des Zugriffs zulässig ist. Im Allgemeinen sind ihm die Adressabbildungsvorschrift sowie die mit den virtuellen Speicherseiten verknüpften Zugriffsbeschränkungen nur teilweise bekannt. Die ihm bekannten Teile sind einerseits durch den Inhalt der im Hauptspeicher liegenden, als „Page Table“ bezeichneten Datenstruktur, andererseits durch den Inhalt der im Bild dargestellten „Sp. Sch.“-Register definiert, welches im Wesentlichen einen Verweis auf die aktuell gültige Page Table enthält. Wenn der Speicherzugriffsakteur mit einem Zugriff auf eine virtuell adressierte Speicherzelle beauftragt wird, den er aufgrund fehlender Information oder wegen eines Zugriffsverbots nicht durchführen kann, fordert er beim Unterbrechungsakteur über den mit der Nummer 4 versehenen Kanal eine Unterbrechung an. Der Unterbrechungsakteur kann den aktuell vom Prozessor ausgeführten Handlungsstrang unterbrechen, um eine vorrangig durchzuführende Aufgabe zu erledigen. Ebenso ist er, nach der Erledigung der

vorrangig zu behandelnden Aufgabe, für die Wiederaufnahme des unterbrochenen Handlungsstrangs zuständig. Neben den beiden genannten Unterbrechungsgründen gibt es weitere. Man unterscheidet üblicherweise zwischen synchronen und asynchronen Unterbrechungen. Synchrone Unterbrechungen sind solche, deren Auftreten eine unmittelbare Konsequenz der Ausführung des aktuellen Befehls darstellt. Asynchrone Unterbrechungen werden hingegen durch externe Ereignisse ausgelöst, deren Auftreten dem Prozessor per Interrupt gemeldet wird. Zwei typische Beispiele für solche externen Ereignisse in einem IBM /360 Rechner sind der Ablauf eines Timers sowie der Abschluss eines Transferauftrags durch einen der Transferprozessoren. Synchrone Unterbrechungen, die auch als Traps bezeichnet werden, können von den entsprechenden Akteuren über die mit Nummern 2 bis 4 versehenen Kanäle beim Unterbrechungsakteur angefordert werden. Über den Kanal mit der Nummer 2 meldet der Operationsakteur Ausnahmesituationen, die bei der Durchführung einer Operationsanweisung auftreten - also beispielsweise Division durch null. Auch der Separator kann über den Kanal mit der Nummer 3 synchrone Unterbrechungen anfordern. Er tut dies beispielsweise dann, wenn durch den zur Ausführung anstehenden Befehl das Auslösen eines Traps explizit verlangt wird oder wenn der Prozessor sich im nicht privilegierten Ausführungsmodus befindet und vom Anweisungsholer ein privilegierter Befehl geholt wurde. Der aktuelle Betriebsmodus des Prozessors ist in dem im Bild dargestellten Speicher mit der Bezeichnung „Priv.“ gespeichert. Wie der Unterbrechungsakteur mit eintreffenden Unterbrechungsanforderungen verfährt, hängt im Allgemeinen von der Belegung des im Bild mit „Unt. Einst.“ bezeichneten Speichers ab. Wenn der Unterbrechungsakteur den Bedarf für eine Unterbrechung des aktuellen Handlungsstrangs feststellt, tauscht er die Inhalte der im Bild grau hinterlegten Speicher aus. Ein solcher Satz von Belegungen für die grau hinterlegten Speicher wird gelegentlich auch als Maschinenkontext bezeichnet. Der Austausch des aktuellen Maschinenkontextes beinhaltet das Retten des alten Kontextes für die spätere Wiederherstellung sowie das Einsetzen eines neuen Kontextes. Da ein infolge einer Unterbrechung zur Ausführung gekommener Handlungsstrang zur Erledigung einer noch dringlicheren Aufgabe erneut unterbrochen werden kann, ist ein Stack für die geretteten Maschinenkontexte erforderlich. Der Speicherort für die Daten dieses Stacks ist der Hauptspeicher, ein Teil der Daten dieses Stacks wird üblicherweise in dem ohnehin zur Unterstützung von Unterprogrammen vorhandenen Stack-Speicherbereich untergebracht. Die Bestimmung der Werte für den neu eingesetzten Kontext geschieht üblicherweise mit Hilfe von Unterbrechungsvektortabellen, die im Hauptspeicher abgelegt sind. In diesen Tabellen ist für jeden Unterbrechungsgrund zumindest ein Startwert für den Programmzähler hinterlegt. Der nach der Unterbrechung herrschende Betriebsmodus wird im Allgemeinen auf privilegiert eingestellt, und die Speicherschutzeinstellungen werden so verändert, dass der Anweisungsholakteur auf die Maschinenanweisungen der Unterbrechungsroutine zugreifen kann. Bei einer asynchronen Unterbrechung werden üblicherweise auch die Unterbrechungseinstellungen neu bestimmt, und zwar so, dass zumindest diejenige Unterbrechung, die den aktuellen Kontextwechsel verursacht hat, nicht zu einer erneuten Unterbrechung führen kann. Die genaue Art der Bestimmung der Belegungen für die neuen Kontextwerte hängt jedoch vom konkreten Prozessortyp ab. Im Fall eines Prozessors eines IBM /360 Systems enthält die Unterbrechungsvektortabelle nicht nur die neuen Startwerte für den PC, sondern neue Werte für sämtliche zum Maschinenkontext gehörende Register.

2.1.1.2 Implementierungstechnik für VMM-Software

Die zweite von Popek und Goldberg in ihrer Definition formulierte Forderung an die VMM-Software verlangt, dass die Ausführung einer statistisch dominanten Teilmenge, der durch einen virtuellen Prozessor ausgeführten Maschinenbefehle, direkt durch den realen Prozessor erfolgen muss und keine Intervention durch den Virtual Machine Monitor erfordern darf. Die VMM-

Software der damaligen Zeit war untrennbar mit der hier beschriebenen Implementierungstechnik verbunden, die diese Forderung automatisch garantiert. Die grundsätzliche Vorgehensweise bei dieser Art der Implementierung von VMM Software beruht im Wesentlichen auf drei Prinzipien, die im Folgenden grob umrissen werden:

I) Virtualisierung des Hauptspeichers

Für jede virtuelle Maschine wird ein eigener virtueller Adressraum verwaltet, dessen Inhalt während der Simulation stets exakt mit demjenigen Inhalt übereinstimmt, der bei Ausführung der gleichen Software im Hauptspeicher einer realen, gleich ausgestatteten Maschine zu finden wäre. In Zusammenhang mit diesem Adressraum für den „virtuellen Hauptspeicher“ stehen zwei Speicherabbildungen. Diese beiden Speicherabbildungen sowie die drei daran beteiligten Adressräume sind in Bild 4 dargestellt. Jeder Adressraum ist durch ein Rechteck mit Koordinatenachse dargestellt. Die genutzten Teile des jeweiligen Adressraums sind weiß, die ungenutzen grau hinterlegt. Der gerade erwähnte, zur Simulation des Hauptspeichers der virtuellen Maschine dienende Adressraum ist in der Bildmitte dargestellt. Links ist der physische Adressraum der

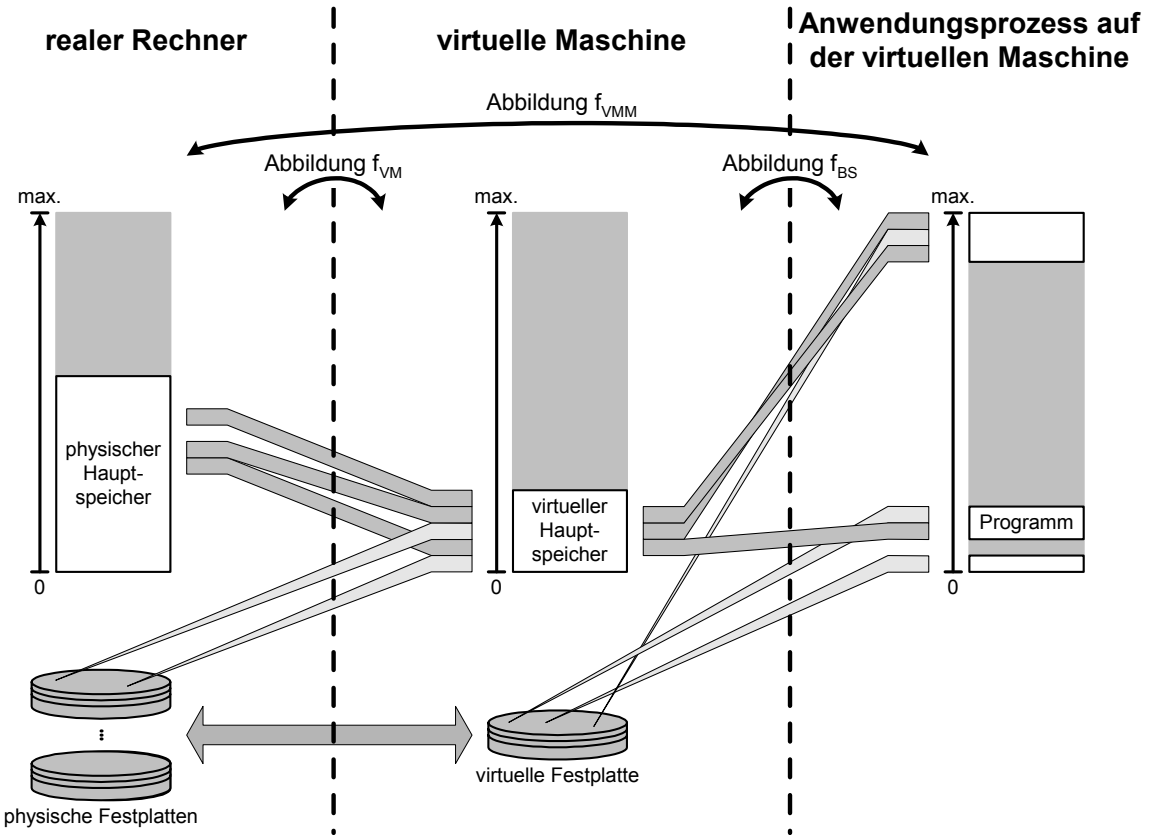


Bild 4: Abbildung von Adressen einer virtuellen Maschine

realen Maschine dargestellt. Falls die auf der virtuellen Maschine installierte Software selbst wieder die durch den Prozessor angebotene Adressvirtualisierung nutzt, ist der für die aktuelle Befehlsausführung gültige Adressraum im Allgemeinen von dem in der Bildmitte dargestellten Adressraum verschieden. Der rechte Adressraum steht stellvertretend für einen solchen Adressraum. Bei der Darstellung in Bild 4 wurde angenommen, dass auf der virtuellen Maschine ein Betriebssystem installiert sei und dass der rechts dargestellte Adressraum der einem Anwendungsprogramm zur Verfügung gestellte sei. Außerdem wurde angenommen, dass der in einem realen Rechner vorhandene Hauptspeicher im physischen Adressraum einen zusammenhängen-

den Adressbereich belegt, der bei Adresse 0 beginnt. In dem zur Simulation des Hauptspeichers der virtuellen Maschine verwendeten Adressraum wird daher ebenfalls nur ein zusammenhängender, ab Adresse 0 beginnender Adressblock genutzt. Die Abbildung des Hauptspeichers der virtuellen Maschine auf die realen Ressourcen ist im Bild mit „f_{VM}“ bezeichnet. Die im Bild mit „f_{BS}“ bezeichnete Abbildung ist die aktuell von der auf der virtuellen Maschine installierten Software eingestellte Speicherabbildung. Prinzipiell kann bei beiden Abbildungen auf Hintergrundspeicher zurückgegriffen werden, um eine Vergrößerung virtueller Adressräume über die Grenzen des physischen/virtuellen Hauptspeichers hinaus zu ermöglichen. Normalerweise ist es jedoch wesentlich effizienter dem virtuellen Rechner gleich ausreichend virtuellen Hauptspeicher zur Verfügung zu stellen, um den Rückgriff auf Hintergrundspeicher durch das Betriebssystem des virtuellen Rechners zu vermeiden. Dies liegt einerseits daran, dass der Zugriff auf Hintergrundspeicher durch virtuelle Rechner nur indirekt über virtuelle Geräte erfolgen kann, für die bei jedem Zugriff Simulationsaufwand erforderlich wird. Andererseits können die zur VMM-Software gehörenden Maschinenbefehle direkt ausgeführt werden, während die Ausführung von Befehlen der virtuellen Maschine der Simulation unterliegt, was speziell bei Programmteilen, die privilegierte Maschinenbefehle beinhalten, zu besonders hohem Zusatzaufwand führt. Im Bild ist außerdem die für die folgenden Betrachtungen wichtige Abbildung „f_{VMM}“ eingezeichnet, die den direkten Zusammenhang zwischen dem Inhalt des Anwendungsadressraums und den physischen Ressourcen der realen Maschine herstellt. Sie ergibt sich aus der Verkettung der beiden bereits besprochenen Abbildungen „f_{VM}“ und „f_{BS}“.

II) Virtualisierung des Prozessors

Die zur Simulation notwendige Ausführung des Programms der virtuellen Maschine erfolgt für alle nicht privilegierten Befehle direkt durch den realen Prozessor. Dazu wird die Adressvirtualisierung der realen Maschine so eingestellt, dass die tatsächlich geleistete Adressumsetzung der Abbildung „f_{VMM}“ entspricht. Vor Beginn der Ausführung von Maschinenbefehlen, die zum Programm der virtuellen Maschine gehören, wird der Prozessor stets in den nicht privilegierten Modus geschaltet. Daher führt die Ausführung von privilegierten Maschinenbefehlen, die im Programm der virtuellen Maschine vorkommen, bei der Simulation immer zu einem Trap. Jeder dieser Traps wird durch entsprechende, zur VMM-Software gehörende, Unterbrechungsroutinen behandelt. Die Art der Behandlung dieser Traps hängt vom Betriebsmodus des virtuellen Prozessors ab. Befindet sich der virtuelle Prozessor im nicht privilegierten Betriebsmodus wird, entsprechend der Belegung der Unterbrechungsvektortabelle des virtuellen Prozessors, ein Kontextwechsel simuliert. Falls der virtuelle Prozessor sich im privilegierten Modus befindet, wird die Ausführung des privilegierten Befehls simuliert.

III) Virtualisierung der Peripherie des Zentralprozessors

Bisher wurde die Vorgehensweise bei der Virtualisierung des Zentralprozessors und des Hauptspeichers vorgestellt. In diesem Abschnitt geht es um die Virtualisierung der Peripheriegeräte und der übrigen Teile der Zentraleinheit. Die Gesamtheit dieser Dinge wird im Folgenden als Prozessorperipherie bezeichnet. Durch die Hardware/Softwareschnittstelle des realen Rechners ist auch die Schnittstelle zur Prozessorperipherie des (virtuellen) Rechners festgelegt. Bei der Kommunikation zwischen Zentralprozessor und seiner Peripherie sind grundsätzlich zwei Fälle zu unterscheiden. Einerseits kann der Zentralprozessor Aufträge an die Prozessor-Peripherie erteilen, andererseits kann die Kommunikation auch durch Interrupts von der Umgebung initiiert werden. Die Erteilung eines Auftrags an die Prozessorperipherie ist stets mit der Ausführung eines privilegierten Befehls verbunden. Dem entsprechend geschieht die durch die VMM-Software zu leistende Abbildung der virtuellen auf die physisch vorhandene Prozessorperipherie einerseits während der Simulation von privilegierten Maschinenbefehlen der virtuellen Maschine

und andererseits bei der Behandlung von Interrupts der realen Maschine. Um einen Eindruck von dem Virtualisierungsaufwand zu bekommen, wird nun beispielhaft die Simulation virtueller Peripheriegeräte auf IBM /360 Systemen skizziert:

III a) Simulation privilegierter Befehle
Der Befehlssatz eines IBM /360 Zentralprozessors enthält im Wesentlichen nur einen privilegierten Befehl zur Erteilung von Aufträgen an Peripheriegeräte, den „SIO“ Befehl. Dieser Befehl dient dazu, die Abwicklung eines Transferprogramms durch einen Transferprozessor anzustoßen. Die Parameter dieses Befehls sind: Der Transferprozessor, der das Programm ausführen soll, das zu verwendende Peripheriegerät sowie die Adresse, ab der das Transferprogramm im Hauptspeicher zu finden ist. Zur Ausführung eines in dem Programm der virtuellen Maschine vorkommenden SIO-Befehls muss Folgendes getan werden: Zunächst muss das im virtuellen Hauptspeicher der virtuellen Maschine vorhandene Transferprogramm in ein Transferprogramm für die reale Maschine übersetzt werden. Bei dieser Übersetzung muss sowohl die Abbildung der virtuellen Geräte auf die physischen Geräte als auch die Abbildung des virtuellen Hauptspeichers auf den physischen Hauptspeicher berücksichtigt werden. Da im Allgemeinen mehrere virtuelle Geräte auf ein physisches Gerät abgebildet werden, kann es sein, dass die Ausführung des übersetzten Transferprogramms nicht unmittelbar angestoßen werden kann, weil die zur Ausführung notwendigen physischen Ressourcen aktuell, aufgrund der Ausführung anderer Transferprogramme, belegt sind. Daher müssen entsprechende Puffer für verzögert auszuführende Transferprogramme verwaltet werden.

III b) Behandlung von Interrupts der realen Maschine
Wenn ein Transferprozessor der realen Maschine die Erledigung eines Transferprogramms durch Auslösen eines Interrupts anzeigt, muss zunächst überprüft werden, ob dieses Transferprogramm zur Simulation eines Transferprogramms einer virtuellen Maschine diente. Falls es sich bei dem Transferauftrag um einen solchen handelt, der durch Übersetzung eines durch eine virtuelle Maschine erteilten Transferauftrags entstanden ist, muss die Simulation des Auftretens des entsprechenden Interrupts in dieser virtuellen Maschine eingeplant werden. Die Frage, wann die Simulation eingeplanter Interrupts virtueller Maschinen tatsächlich erfolgen sollte, kann nur unter Berücksichtigung von Wissen über das Zeitverhalten der realen Prozessorperipherie sowie über das von der Software der virtuellen Maschine vorausgesetzte Zeitverhalten sinnvoll beantwortet werden. Sie kann daher hier nicht weiter verfolgt werden. Zur Simulation eines Interrupts der virtuellen Maschine müssen zunächst die Unterbrechungseinstellungen des virtuellen Prozessors dahingehend überprüft werden, ob der Interrupt aktuell zulässig ist. Falls ja, muss ein Kontextwechsel des virtuellen Prozessors simuliert werden.

2.1.1.3 „Virtualizable Architectures“

Die Anwendung der im letzten Abschnitt vorgestellten Implementierungstechnik stellt Anforderungen an die Rechnerhardware. Speziell die vorgestellte Art der Virtualisierung des Prozessors konnte bei einer Vielzahl der Third Generation Architectures nicht angewendet werden. Aber auch die aufgrund der Virtualisierung des I/O Systems veränderten Timing-Eigenschaften der virtuellen Peripheriegräte stellten ein großes Problem dar. Auf einige der nun folgenden Hardwareanforderungen wird auch in [Popek Goldberg 74] hingewiesen.

a) Anforderungen an den Zentralprozessor

Die vorgestellte Art der Virtualisierung basiert darauf, dass die Ausführung aller nicht privilegierten Maschinenbefehle, die in Programmen der virtuellen Maschine vorkommen, direkt

durch den Prozessor erfolgen kann, wenn die Adressvirtualisierungseinstellungen der realen Maschine geeignet gewählt werden. Außerdem setzt dies voraus, dass alle physischen Prozessorregister, deren Inhalt für die Ausführung nicht privilegierter Befehle relevant werden kann, während der Ausführung solcher Befehle den gleichen Inhalt haben wie die entsprechenden „virtuellen Register“ des Prozessors der virtuellen Maschine. Die physischen Register, in denen die Speicherschutzeinstellungen, die Unterbrechungseinstellungen und die Betriebsart gespeichert sind, können aber während der Ausführung von Maschinenbefehlen des Programms der virtuellen Maschine zwangsläufig nicht die gleichen Inhalte haben wie die entsprechenden virtuellen Register. Diese Register dürfen daher nicht als Quelle für Operanden von nicht privilegierten Befehlen vorkommen.
Damit privilegierte Befehle, die im Programm der virtuellen Maschine enthalten sind, entsprechend dem beschriebenen Prinzip simuliert werden können, ist es außerdem erforderlich, dass der Prozessor immer eine Trap-Behandlung einleitet, wenn die Ausführung eines privilegierten Befehls in der Betriebsart „non privileged mode“ verlangt wird.
Die Adressvirtualisierung des physischen Prozessors muss während der Ausführung von Maschinenbefehlen des Programms der virtuellen Maschine so eingestellt werden, dass die tatsächlich geleistete Adressumsetzung der Abbildung „f_{VMM}“ aus Bild 4 entspricht. Die Abbildung „f_{VMM}“ ist jedoch von den aktuellen Adressvirtualisierungseinstellungen der virtuellen Maschine abhängig, welche maßgeblich durch den aktuellen Inhalt der Page Table des virtuellen Prozessors definiert werden. Daher machen Änderungen am Inhalt der aktuell gültigen Page Table des virtuellen Prozessors im Allgemeinen auch Änderungen an der aktuell gültigen Page Table des physischen Prozessors erforderlich. Da die Page Table des virtuellen Prozessors im virtuellen Hauptspeicher liegt, kann ihr Inhalt durch die Ausführung nicht privilegierter Maschinenbefehle verändert werden. Um die stetige Aktualisierung der Adressvirtualisierungseinstellungen des physischen Prozessors durch die VMM-Software gewährleisten zu können, muss die Adressvirtualisierung des physischen Prozessors so eingestellt werden können, dass jegliche Veränderungen an der Page Table des virtuellen Prozessors eine Trap Behandlung auslösen.

b) Anforderungen an die Peripherieankopplung

Die Anforderungen an die Peripherieankopplung können nicht allgemein gefasst werden, da der grundsätzliche Aufbau des Peripheriesystems von Hersteller zu Hersteller stark variierte. Daher wird an dieser Stelle nur auf einige besondere Konstruktionsmerkmale des I/O-Systems der IBM /360 Familie und deren Einfluss auf die Virtualisierbarkeit des I/O-Systems hingewiesen.

Wie bereits erwähnt, verfügten die Rechner der IBM /360 Familie über spezielle Transferprozessoren für die Abwicklung des Datenaustauschs mit der Peripherie. Bei vielen anderen Hardwarearchitekturen war das I/O-System so aufgebaut, dass der Zentralprozessor (zumindest teilweise) auch für die Abwicklung der Kommunikationsprotokolle mit den Peripheriegeräten zuständig war. Dies führt unter anderem zu den folgenden beiden, für die Virtualisierung nachteiligen Konsequenzen: Einerseits muss der Zentralprozessor in diesem Fall wesentlich mehr Interrupts verarbeiten. Andererseits müssen bei der Abwicklung der Kommunikationsprotokolle üblicherweise bestimmte Reaktionszeiten eingehalten werden, woraus sich entsprechende Anforderungen an die Interruptbearbeitungszeiten ergeben. Im Fall eines IBM /360 Rechners übernehmen die Transferprozessoren die in Bezug auf die Reaktionszeiten kritischen Entscheidungen, so dass die Interruptreaktionszeiten des Zentralprozessors bei solchen Entscheidungen irrelevant sind.
Die Konstruktion des I/O-Systems der IBM /360 Familie eröffnete im Gegensatz zu vielen anderen I/O-Systemen die Möglichkeit für eine effiziente Virtualisierung. Der Hauptvorteil dieses I/O-Systems lag darin, dass der Zentralprozessor weitgehend von der Behandlung zeitkritischer Interrupts befreit war. Einerseits reduzierte sich dadurch sowohl die Anzahl der zu behandeln-

den realen Interrupts wie auch die Anzahl der zu simulierenden virtuellen Interrupts. Andererseits reduzierte dies auch den Bedarf für die Umschaltung der Simulation zwischen den verschiedenen virtuellen Maschinen, was eine bessere Nutzung des Zentralprozessor mit sich brachte. Das schwierigste Problem bei der Virtualisierung des I/O-Systems der IBM /360 Rechner war die Übersetzung der virtuellen Transferprogramme in solche für die reale Maschine. Dies war insbesondere deshalb so schwierig, weil die Transferprozessoren grundsätzlich in der Lage waren, den Code des aktuell in Ausführung befindlichen Transferprogramms zu ändern.

2.1.2 Virtuelle Maschinen mit direktem Bezug zu realer Hardware

Im Fall klassischer virtueller Maschinen waren die virtuellen Maschinen stets Duplikate der zur Simulation verwendeten realen Maschine. Auch heute spielt die Nachbildung von Maschinen deren konstituierende Charakteristika auf real existierende Rechnerhardware zurückgehen, noch eine wichtige Rolle. Dabei wird jedoch auch dann von virtuellen Maschinen gesprochen, wenn die konstituierenden Charakteristika der virtuellen Maschine nicht aus der Hardware/ Softwareschnittstelle der zur Simulation eingesetzten Maschine abgeleitet wurden. Da in solchen Fällen die Voraussetzungen für die Anwendung der vorgestellten Implementierungstechnik im Allgemeinen nicht erfüllt sind, beinhaltet Virtualisierungssoftware heute üblicherweise einen „Binary Translator“, der Maschinenprogramme, die durch den virtuellen Prozessor auszuführen sind, in solche für den realen Prozessor übersetzt. Ein weiterer Unterschied heutiger Virtualisierungssoftware besteht darin, dass sie häufig nicht mehr „direkt auf der Hardware“ ausgeführt wird, sondern bereits ein Betriebssystem voraussetzt.

Im Zusammenhang mit dem Ursprung der für eine virtuelle Maschine kennzeichnenden Charakteristika stehen die beiden in der Literatur gebräuchlichen Begriffe „**I**nstruction **S**et **A**rchitecture“ (ISA) und „**A**pplication **B**inary **I**nterface“ (ABI) die jetzt anhand von Bild 5 vorgestellt werden. Der Einsatz von Betriebssystemen ist für heutige programmiert realisierte

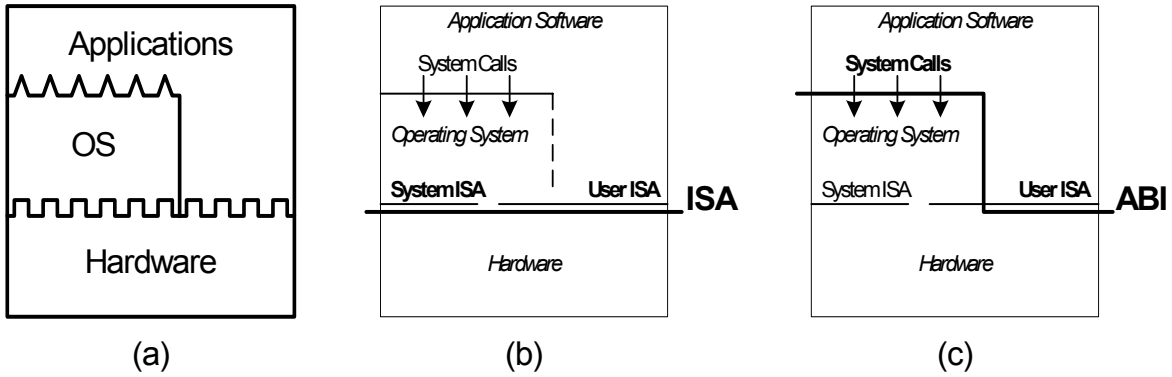


*Bild 5: „**I**nstruction **S**et **A**rchitecture“ und „**A**pplication **B**inary **I**nterface“ nach [Smith Nair 04]*

Systeme kennzeichnend. Damit ergibt sich automatisch die im Bild 5 (a) dargestellte Schichtung. Die Darstellung in Bild 5 (a) wird in Bild 5 (b) und Bild 5 (c) zur Veranschaulichung der beiden Begriffe wieder aufgegriffen.
Der Begriff „Instruction Set Architecture“ stammt aus den 60er Jahren und ist gleichbedeutend mit dem Begriff der „Hardware/Softwareschnittstelle“ einer Maschine. Er ist ein Sammelbegriff für all diejenigen Eigenschaften einer realen Maschine, die durch die Konstruktion der Maschine festgelegt und für die geleistete Informationsverarbeitung relevant sind. Das durch die „Instruction Set Architecture“ festgelegte Modell der realen Maschine umfasst insbesondere Modelle aller im System vorhandenen Prozessoren, deren Programm unmittelbar durch einen

Systemprogrammierer verändert werden kann. Neben diesen Prozessormodellen, die die Bedeutung der Abwicklung der zugehörigen Maschinenbefehle festlegen, ist auch ein Modell, das die Synchronisation der verschiedenen Prozessoren festlegt, wesentlicher Bestandteil der „Instruction Set Architecture“.
Ebenso wie die „Instruction Set Architecture“ ein Maschinenmodell definiert, welches alle aus Sicht eines Systemprogrammierers relevanten Eigenschaften einer realen Maschine umfasst, definiert auch das „Application Binary Interface“ ein Maschinenmodell. Das durch das „Application Binary Interface“ definierte Maschinenmodell ist jedoch kein Modell der realen Maschine, sondern das Modell derjenigen abstrakten Maschine, die für einen Anwendungsprogrammierer maßgeblich ist. Die Eigenschaften dieser abstrakten Maschine hängen nur teilweise von den Eigenschaften der realen Maschine ab und werden zu einem wesentlichen Teil durch das Betriebssystem festgelegt. Insbesondere wird der Prozessor während der Ausführung von Maschinenbefehlen, die dem Code von Anwendungsprogrammen zuzurechnen sind, stets im nicht privilegierten Modus betrieben, so dass der Zugriff auf Peripheriegeräte nur über Betriebssystemaufrufe und nicht mehr „direkt“ durch Anwendung entsprechender privilegierter Befehle (System ISA) möglich ist.
In Anlehnung an [Smith Nair 04] werden virtuelle Maschinen, deren konstituierende Charakteristika aus einer Hardware/Softwareschnittstelle abgeleitet wurden, im Folgenden als „System VMs“ und virtuelle Maschinen, deren konstituierende Charakteristika aus einem „Application Binary Interface“ abgeleitet wurden, als „Process VMs“ bezeichnet. In diesem Sinne sind klassische virtuelle Maschinen als System VMs zu bezeichnen.

Das Spektrum an virtuellen Maschinen, deren konstituierende Charakteristika auf real existierende Rechnerhardware zurückgehen, ist so groß, dass eine umfassende Darstellung an dieser Stelle nicht möglich ist. Daher wird dieses Spektrum im Folgenden nur grob umrissen. In den Bidern 6 und 7 sind fünf typische Fälle der Anwendung von Virtualisierungssoftware veranschaulicht. Die Art der Darstellung hat dabei die in Bild 5 (a) verwendete zum Vorbild. Die Aufgabe der jeweiligen Virtualisierungssoftware wird durch die grau hinterlegten „Bausteine“

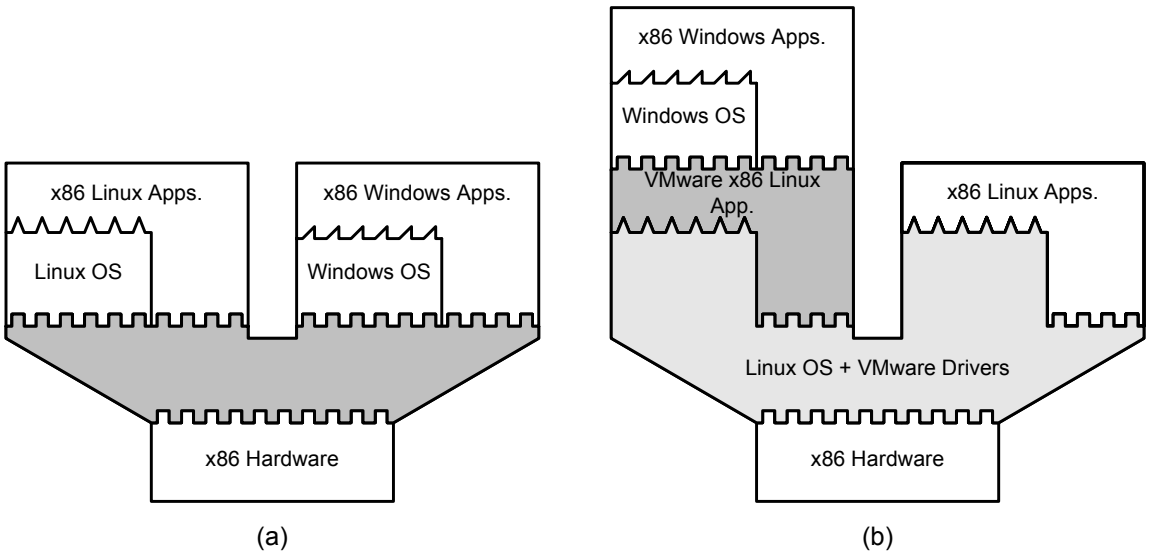


Bild 6: Beispiele für den Einsatz von Virtualisierungssoftware (nach [Smith Nair 04])

repräsentiert. Der Zweck der in Bild 6 (a) veranschaulichten Virtualisierungssoftware ist mit dem von klassischer Virtual Machine Monitor Software identisch und besteht in der Vervielfachung der Hardware/Softwareschnittstelle der realen Maschine. Die verwendete Bezeichnung „x86 Hardware“ soll andeuten, dass die vorausgesetzte Hardware PC Hardware sein soll, die

mit Prozessoren ausgestattet ist, deren Befehlssatz zu dem der „x86“ Prozessoren der Firma Intel kompatibel ist. Auf den bereitgestellten virtuellen Maschinen können zeitgleich verschiedene Betriebssysteme betrieben und die jeweils passenden Anwendungsprogramme genutzt werden. Beispielsweise könnten das die für die Darstellung angenommenen „x86“ Versionen der Betriebssysteme Linux und Windows sowie die für das jeweilige Betriebssystem verfügbaren „x86“ Versionen von Anwendungsprogrammen sein. Der in Bild 6 (a) dargestellte Fall deckt sich im Unterschied zu den anderen drei in Bild 6 dargestellten Fällen mit keiner real existierenden Implementierung von Virtualisierungssoftware. Es gibt jedoch Virtualisierungssoftware die System VMs mit „x86“-ISA bereitstellt und unter anderem auch eine Maschine mit „x86“-ISA voraussetzt. Ein Beispiel hierfür ist die in Bild 6 (b) veranschaulichte Software mit der Produktbezeichnung „VMware“. Diese Software setzt neben einer Maschine mit „x86“-ISA auch voraus, dass auf dieser Maschine bereits ein Betriebssystem inklusive Treibern für sämtliche im System vorhandene Hardware installiert ist. Die zum Betrieb von VMware benötigten Voraussetzungen decken sich auch nicht mit denen, die zum Betrieb „normaler“ Anwendungsprogramme notwendig sind, da die Installation von VMware selbst weitere Treiber zur Erweiterung des vorhandenen Betriebssystems mit sich bringt. Klassische VMM-Software wird direkt auf der Hardware ausgeführt und umfasst insbesondere auch Treiber zur Ansteuerung der physisch vorhandenen Hardware. Angesichts der Hardwarevielfalt, die im Bereich der „x86“ PC-Hardware vorzufinden ist, war diese Vorgehensweise im Fall von VMware praktisch unmöglich. Stattdessen wurde dieses Problem hier dadurch umgangen, dass die Hardware nicht direkt, sondern indirekt über das vorhandene Betriebssystem angesteuert wird.

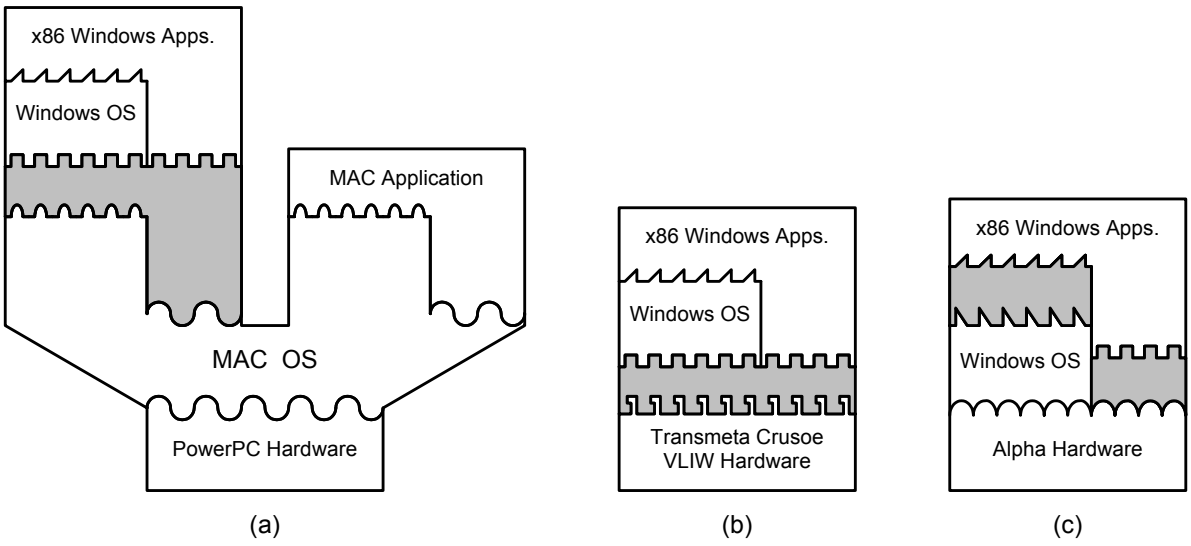


Bild 7: Beispiele für den Einsatz von Virtualisierungssoftware (nach [Smith Nair 04])

Bild 7 (a) zeigt Virtualisierungssoftware die System VMs mit „x86“-ISA bereitstellt und eine MAC OS Process VM voraussetzt. Das Beispiel geht auf das Softwareprodukt „Virtual PC“ zurück. Ähnlich wie im Fall klassischer virtueller Maschinen liegt auch hier die Motivation für die Entwicklung dieser Software in der Herstellung von Softwarekompatibilität. Die Anwendung der Implementierungstechnik klassischer VMM Software ist aufgrund der unterschiedlichen Befehlssätze der PowerPC und x86 Prozessoren von vornherein ausgeschlossen. Um dennoch eine akzeptable Simulationsgeschwindigkeit zu erreichen, beinhaltet die Virtual PC Software daher einen „Binary Translator“, der x86 Maschinenprogramme in solche Maschinenprogramme übersetzen kann, die für den Power PC Prozessor geeignet sind.
Bild 7 (b) soll nun zum Anlass genommen werden auf ein Anwendungsgebiet von Virtualisie-

rungssoftware aufmerksam zu machen, das nicht durch den Bedarf nach Kompatibilität zwischen existierender Software und existierender Hardware, sondern vielmehr durch die Entwicklung neuer Prozessorhardware begründet ist. Derartige virtuelle Maschinen werden von [Smith Nair 04] auch als „Co-Designed VMs“ bezeichnet, da die Entwicklung der Virtualisierungssoftware untrennbar mit der Entwicklung der Prozessorhardware verbunden ist. Es ist für Systeme, bei deren Konstruktion das Co-Designed VM Konzept verfolgt wurde, kennzeichnend, dass die Virtualisierungssoftware die einzige Software ist, die unmittelbar „auf der Hardware“ ausgeführt wird. Die gesamte übrige Software setzt einen anderen Prozessor voraus, der von der Virtualisierungssoftware simuliert wird. Im dargestellten Fall des Transmeta Crusoe Prozessors wurde die Co-Designed VM im Hinblick auf die Bereitstellung eines virtuellen x86 Prozessors entworfen. Auf solchen mit einem Transmeta Crusoe Prozessor ausgestatteten Maschinen können alle ursprünglich für die „x86“ Hardware entworfenen Betriebssysteme zusammen mit den dafür entwickelten Anwendungsprogrammen unverändert eingesetzt werden. Ein weiterer Prozessor, bei dem das Co-Designed VM Konzept verfolgt wurde, ist der Daisy Prozessor von IBM, auf dem ein virtueller PowerPC Prozessor simuliert werden kann. Die zusätzlich eingeführte Virtualisierungsschicht hat im Wesentlichen zwei Vorteile: Beim Entwurf neuer Prozessorhardware ist die Rückwärtskompatibilität zu Vorgängermodellen oft sehr wichtig. Diese Kompatibilitätsforderung verursacht oft großen Mehraufwand bei der Entwicklung neuer Prozessorhardware und erschwert grundlegende Änderungen. Im Fall von Co-Designed VMs kann die zur Erfüllung dieser Kompatibilitätsforderung notwendige Funktionalität in der Virtualisierungssoftware implementiert werden. Der zweite Vorteil dieses Konzeptes besteht in der großen Vielfalt von Optimierungen, die relativ günstig in der Virtualisierungssoftware implementiert werden können. Bei den als Beispiel angeführten Prozessoren sind die Befehlssätze des simulierten Prozessors und physischen Prozessors völlig verschieden. Die Virtualisierungssoftware enthält in beiden Fällen einen Binary Translator der optimierten Code erzeugt. Die Optimierungen sind in den beiden genannten Fällen nicht statisch, sondern berücksichtigen das aktuelle „Kontrollflussverhalten“ der durch den simulierten Prozessor auszuführenden Maschinenprogramme. Da zur Speicherung der Übersetzungsergebnisse und zur Durchführung der Optimierungen eine beträchtliche Menge an Speicher erforderlich ist, wird ein Teil des Hauptspeichers speziell für diesen Zweck reserviert. Während die Optimierungen im Fall des Daisy Prozessors allein auf die Maximierung der Rechenleistung abzielen, spielt im Fall des Crusoe Prozessors die Senkung des Stromverbrauchs eine ebenso wichtige Rolle.

Bild 7 (c) veranschaulicht die Aufgabenstellung für die Virtualisierungssoftware „FX!32“. In diesem Fall ging es darum, Windows Anwendungsprogramme, die für „x86“-kompatible 32-Bit Prozessoren übersetzt worden waren, auf Rechnerhardware benutzen zu können, die mit 64-Bit Alpha Prozessoren ausgerüstet war. Auf den mit den Alpha Prozessoren ausgerüsteten Maschinen wurde dabei die gleiche Version des Betriebssystems Windows NT betrieben, wie diejenige die von den betreffenden Anwendungsprogrammen vorausgesetzt wurde. Dadurch reduzierte sich die Aufgabenstellung im Wesentlichen auf die Virtualisierung des „x86“ Prozessors. Die FX!32 Software umfasst hierzu einen optimierenden Binary Translator, der bei den Optimierungen das „Kontrollflussverhalten“ der zu übersetzenden „x86“ Maschinenprogramme berücksichtigt. Die von diesem Binary Translator erzeugten Maschinenprogramme für den Alpha Prozessor werden außerdem dauerhaft auf Festplatte gesichert, so dass die Übersetzungsergebnisse auch nach einem Neustart des mit der FX!32 Software ausgestatteten Systems gleich wieder zur Verfügung stehen.

2.1.3 High Level Language VMs

Wie bereits zu Beginn von Abschnitt 2.1 erwähnt, hat man sehr bald erkannt, dass man das Ziel der plattformübergreifenden Verwendbarkeit von Software viel leichter erreichen kann, wenn man dies gleich bei der Entwicklung der Software berücksichtigt und diese Software für eine abstrakte Maschine entwirft, die man leicht auf allen in Frage kommenden realen Maschinen „simulieren“ kann. Trotz der Tatsache, dass die konstituierenden Charakteristika solcher Maschinen meist nicht von realen Maschinen abgeleitet sind, werden solche Maschinen heute ebenfalls als virtuelle Maschinen bezeichnet. Da die Entwicklung solcher Virtueller Maschinen in der Vergangenheit nahezu immer mit der Entwicklung einer bestimmten (höheren) Programmiersprache in Zusammenhang stand, werden solche Maschinen in der Literatur auch als „High Level Language VMs“ bezeichnet.

Inzwischen gibt es unzählige Sprachimplementierungen, die den High Level Language VM Ansatz verfolgen. Wie sich im Verlauf der nun folgenden Darstellung noch zeigen wird, weist die Java VM, die in Abschnitt 2.2 noch detailliert vorgestellt wird, viele für moderne VMs typische Konstruktionsmerkmale auf. Daher beschränkt sich dieser Abschnitt auf die Darstellung einiger grundsätzlicher Überlegungen zur Konzeption moderner „High Level Language VMs“ und geht gar nicht auf Details einzelner VMs ein.

Eine frühe, weit verbreitete Sprachimplementierung bei der der High Level Language VM Ansatz verfolgt wurde, war das in den 70’er Jahren an der **U**niversity of **C**alifornia **S**an **D**iego entwickelte UCSD-PASCAL System, das unter anderem auf den damals neu aufgekommenen Apple Computern zum Einsatz kam. Die zugehörige Virtuelle Maschine wird als P-Maschine und der „HLL VM Code“ für diese Maschine wird als P-Code bezeichnet. Die P-Maschine ist eine Stack Maschine, deren Eigenschaften sich an der damaligen 16-Bit Prozessorhardware orientieren. Ihr liegt die Vorstellung eines linearen 16-Bit Adressraums zugrunde, und ihr Befehlssatz umfasst nur Arithmetikbefehle für Ganzzahlarithmetik. Die Programmdarstellung in Form von P-Code zeichnet sich durch seine besondere „Kompaktheit“ aus. Zur Ausführung der Programme auf der Apple Hardware wurde ein entsprechender Interpreter verwendet.

Während die Eigenschaften des P-Code[5] stark durch Hardware-Eigenschaften geprägt waren, wird das Austauschformat heutiger High Level Language Implementierungen oft wesentlich durch die Konzepte der entsprechenden Programmiersprachen beeinflusst. So gibt es große Unterschiede zwischen den Maschinenmodellen, die den Austauschformaten imperativer, funktionaler, prädikativer bzw. objektorientierter Programmiersprachen zugrunde liegen. Dass die Konzepte der betreffenden Programmiersprache heute einen größeren Einfluss auf das Austauschformat haben, liegt insbesondere auch daran, dass Programme heute nicht mehr „monolitisch“ sind, sondern aus einer Menge von „Inkrementen“ unterschiedlicher Hersteller bestehen. So bestehen Java Programme beispielsweise aus einer Menge von „Class-Files“ und Programme, die für die Microsoft „.Net“ Laufzeitumgebung entwickelt wurden, aus einer Menge von „Assemblies“.

Die Codierung von Ablaufbeschreibungen geschieht, ebenso wie im Fall des P-Codes, auch heute noch oft in Form von Bytecode für eine Stack Maschine. Im Unterschied zu damals werden diese in Form von Bytecode vorliegenden Ablaufbeschreibungen heute jedoch meist gar

5. Interessanterweise sind sowohl der „P-Code“ als auch die „Slim Binaries“ am Lehrstuhl von Nikolaus Wirth entstanden. Überdies war der P-Code ursprünglich gar nicht zur Verwendung als Austauschformat, sondern vielmehr als Intermediate Representation für den an der ETH Zürich entwickelten PASCAL Compiler entworfen worden.

nicht mehr direkt von einem Interpreter ausgewertet, sondern dienen häufig als Eingabe für einen Compiler, der daraus eine andere Programmdarstellung generiert, welche zur Ausführung herangezogen wird. Daher ist die Entscheidung, Ablaufbeschreibungen in Form von Bytecode zu codieren, heute in viel geringerem Maße dadurch motiviert, diesen Bytecode „direkt“ ausführen zu können. Für die Wahl dieser Codierungsform ist heute oft ausschließlich die Tatsache ausschlaggebend, dass er eine kompakte Darstellung von Ablaufbeschreibungen ermöglicht. Dass dieser Bytecode nicht mehr direkt interpretiert wird, sondern als Eingabe für einen weiteren Übersetzungsschritt dient, hat auch Einfluss auf die Compileroptimierungen, die bei der Erzeugung dieses Bytecodes angewendet werden. Vor diesem Hintergrund ist es nämlich schädlich, wenn der Bytecode bei der Erzeugung „zu stark“ optimiert wird. Viel wichtiger ist es hier, dass bei dem späteren (zweiten) Übersetzungsvorgang noch ausreichend Information über das Quellprogramm vorliegt, um eine sinnvolle Optimierung durchführen zu können. Es gibt daher auch Ansätze, andere Codierungsformen für Ablaufbeschreibungen zu verwenden, die als Eingabe für den zweiten Übersetzungsvorgang besser geeignet sind. Ein interessanter Vorschlag zu diesem Thema sind die so genannten „Slim Binaries“ [Franz Kistler 97] [Kistler Franz 97]. Die Programmdarstellung in Form von „Slim Binaries“ soll dabei doppelt so kompakt sein wie P-Code oder Java-Bytecode und sich überdies besser als Eingabe für den zweiten Übersetzungsschritt eignen.

Trend zu objektorientierten „safe language environments“

In letzter Zeit hat die Bedeutung so genannter „safe language environments“ stetig zugenommen. Solche „safe language environments“ lassen eine höhere Produktivität bei der Softwareentwicklung zu, weil ihre Konstruktion das Auftreten einer bestimmten Klasse von besonders schwierig aufzuspürenden, typischen Programmierfehlern verhindert. Diese Fehler werden in der Literatur gelegentlich auch als „memory errors“ bezeichnet. Typische Beispiele für solche Fehler sind:

- Zugriffe auf nicht existierende Array Elemente.
- Zugriffe auf Speicherbereiche, die inzwischen wieder freigegeben wurden.
- Durchführung von Prozeduraufrufen mit Parametern, die nicht zur Signatur der betreffenden Prozedur konform sind.

Um zumindest einen Teil dieser Fehler vermeiden zu können, wurden in der Vergangenheit Programmiersprachen mit strengen Typsystemen entwickelt. Solche Typsysteme zwingen den Programmierer dazu, dem Compiler, der zur Übersetzung der Quellprogramme verwendet wird, ausreichend Information zur Verfügung zu stellen, um solche Fehler bei der Übersetzung aufdecken zu können. Das Vorhaben, alle möglichen Quellen solcher Fehler mit Hilfe eines entsprechend strengen Typsystems ausschließen zu wollen, zieht jedoch das Problem nach sich, dass dadurch die Ausdrucksmächtigkeit der betreffenden Sprachen stark eingeschränkt wird. Des Weiteren steht diese Vorgehensweise mit dem bereits angesprochenen Bedarf in Konflikt, Programme erst unmittelbar vor dem Start aus „Inkrementen“ unterschiedlicher Hersteller zusammensetzen zu können.

Mit den gerade angesprochenen „(memory) safe language environments“ wird daher ein anderer Weg verfolgt. Sie nutzen eine Kombination unterschiedlicher Mechanismen, um das Auftreten von „memory errors“ während der Ausführung von Programmen auszuschließen. Ihr Design ist daher durch folgende Überlegungen geprägt:

(1) Das Problem des Zugriffs auf Speicherbereiche, die inzwischen wieder freigegeben wurden, wird hier dadurch verhindert, dass die High Level Language VMs eine „manuelle

Speicherverwaltung“ nicht zulassen. Stattdessen stellen solche VMs eine automatische Speicherverwaltung zur Verfügung, mit der zugleich auch das Problem der „memory leaks“ umgangen wird. Die automatische Speicherverwaltung durch eine solche VM beruht typischerweise auf dem Prinzip der Garbage Collection und stellt sicher, dass einmal angeforderter Speicher erst dann wieder freigegeben wird, wenn nachgewiesen werden kann, dass im weiteren Verlauf der Programmausführung nicht mehr auf diesen Speicher zugegriffen wird.

(2) Das Problem des Zugriffs auf nicht existierende Array Elemente wird durch entsprechende Überprüfungen der bei Array Zugriffen verwendeten Indexwerte sichergestellt. Die Frage, ob bei jedem im Programmcode formulierten Arrayzugriff stets nur gültige Indexwerte verwendet werden, kann im Allgemeinen nicht allein durch eine statische Analyse des betreffenden Programmcodes entschieden werden. Daher sind entsprechende Überprüfungen während der Programmausführung oft unumgänglich.

(3) Um die Realisierung typischer dynamischer Datenstrukturen wie Listen, Bäume, etc. formulieren zu können, ist es praktisch unumgänglich, dass die betreffenden Programmiersprachen „Zeiger“-Datentypen zulassen. Um bei Zugriffen „über Zeiger“ die Einhaltung der „memory safety“ und der übrigen durch das Typsystem definierten Regeln sicherstellen zu können, bedarf es in diesem Fall ebenfalls detaillierter Überprüfungen. Um eine akzeptable Performance bei der Programmausführung erzielen zu können, ist es von entscheidender Bedeutung, die Einhaltung der entsprechenden Regeln (nahezu) vollständig anhand von statischen Analysen des auszuführenden Programmcodes überprüfen zu können.

Mit dem Begriff „(memory) safe language environment“ wird daher ein System aus verschiedenen Komponenten bezeichnet. Bild 8 zeigt die Komponenten des „Java language environ-

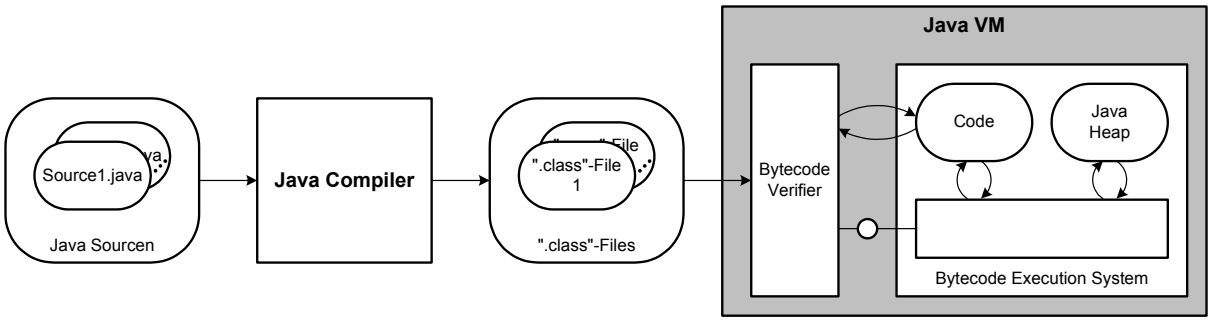


Bild 8: „Java language environment“

ments“. Die links dargestellten Java Sourcen werden mit Hilfe eines Java Compilers in „.class“-Files übersetzt. Der Java Compiler erzeugt dabei typischerweise zu jeder in den Sourcen definierten Java Klasse genau ein „.class“-File. Die spätere Ausführung des in den „.class“ Files enthaltenen Codes erfolgt durch die rechts dargestellte, grau hinterlegte Java VM. Bevor der die in den „.class“-Files enthaltene Bytecode zur Ausführung an das „Bytecode Execution System“ übergeben wird, wird er zunächst vom Bytecode Verifier verifiziert.
Im Rahmen der Bytecode Verifizierung werden die angesprochenen statischen Analysen des Bytecodes durchgeführt, die es ermöglichen, dass bei der späteren Ausführung des Bytecodes weitgehend auf detaillierte Überprüfungen verzichtet werden kann, ohne die Verletzung der „memory safety“ oder der durch das Typsystem festgelegten Regeln befürchten zu müssen. Speziell der Punkt 3 der obigen Auflistung macht eine detaillierte Datenflussanalyse erforderlich. Diese Überprüfung durch den Bytecode Verifier geschieht typischerweise derart, dass er

überprüft, ob jede „Zeiger"-Variable auf zulässige Weise initialisiert wird und ob der Wert dieser Variablen nach der Initialisierung nur durch andere zulässige Werte ersetzt werden kann. Dass der Bytecode Verifier die korrekte Nutzung von „Zeiger"-Variablen überhaupt durch eine statische Programmcodeanalyse überprüfen kann, ist dabei keineswegs selbstverständlich, sondern ist letztlich nur deswegen möglich, weil Java keine „Zeiger"-Arithmetik zulässt und außerdem kein Sprachkonstrukt zur Definition von „untagged variant records" oder „unions" zur Verfügung stellt.

„verifiable Code"

Bytecode nach Art des „Java"-Bytecode, bei dem die gerade beschriebene Überprüfung der korrekten Nutzung von „Zeiger" Variablen durch eine statische Code Analyse möglich ist, wird in der Literatur auch als „verifiable code" bezeichnet. An dieser Stelle soll darauf hingewiesen werden, dass nicht jede Quellsprache (sinnvoll) in „verifiable Code" übersetzt werden kann. Beispielsweise wird dies im Fall der Programmiersprachen „C" oder „C++" durch die Möglichkeit zur Formulierung von Zeigerarithmetik verhindert.

„.NET Framework" als Universalplattform

Die Java VM steht, wie für High Level Language VMs typisch, im direkten Zusammenhang mit der Entwicklung einer ganz bestimmten High Level Language. Obwohl es prinzipiell auch möglich ist, die Java VM zur Ausführung von Programmen einzusetzen, die in anderen Quellsprachen als Java formuliert wurden, eignet sie sich nicht besonders gut zu diesem Zweck.[6] Ferner eignet sie sich prinzipiell nicht zur Ausführung von Programmen, die in Programmiersprachen formuliert wurden, welche nicht in „verifiable Code" übersetzt werden können. Die im Zuge der Entwicklung des „.NET Frameworks" entwickelte High Level Language VM, die unter dem Namen „**C**ommon **L**anguage **R**untime (CLR)" bekannt ist, wurde hingegen speziell im Hinblick auf die Schaffung einer Universalplattform entworfen (siehe [ECMA 01], [Gough 01]). Bei dem in den bereits angesprochenen Assemblies enthaltenen Bytecode handelt sich um CIL-Code. CIL ist dabei ein Akronym für **C**ommon **I**ntermediate **L**anguage. Das CLR kann, ebenso wie die Java VM, als Basis für ein „safe language environment" eingesetzt werden. Es beinhaltet hierzu einen entsprechenden Verifier, einen Garbage Collector und ein sehr leistungsfähiges „**V**irtual **E**xecution **S**ystem (VES)", das über die notwendigen Eigenschaften verfügt, um eine „memory/type safe"-Form der Programmausführung zu gewährleisten. Darüber hinaus ermöglicht das CLR aber auch die Ausführung von „unverifiable Code". Inzwischen wurden für eine ganze Reihe unterschiedlicher Programmiersprachen Compiler entwickelt, die in der Lage sind, Code zur Ausführung durch das CLR zu erzeugen. Allen voran ist hier die Programmiersprache „C#" zu nennen, die es erlaubt, alle Möglichkeiten des CLR voll auszuschöpfen. Weitere Programmiersprachen sind: „COBOL", „C", „Pascal", „Fortran", „Eiffel", „Oberon", „Component Pascal".

Zum Begriff „language based system"

Die Fortschritte auf dem Gebiet der Programmiersprachen und die damit in Zusammenhang stehende Entwicklung von „safe language environments" führten in der Folge zu der Idee, die von solchen VMs garantierten Zugriffsbeschränkungen zum Zwecke der Abschottung verschiedener Systemkomponenten innerhalb einer solchen VM auszunutzen. Systeme, bei deren Entwurf

6. Aufgrund der Tatsache, dass praktisch für jede Hardwareplattform eine leistungsfähige Java VM Implementierung existiert, gibt es dennoch eine Reihe von Compilern für andere Programmiersprachen, die genau diesen Ansatz verfolgen.

diese Idee aufgegriffen wurde, werden in der Literatur häufig als „language based systems“ bezeichnet. Führt man diesen Gedanken konsequent weiter, so stellt sich die Frage, ob man die von solchen VMs garantierten Zugriffsbeschränkungen nicht als Grundlage für die wechselseitige Abschottung von Betriebssystemprozessen verwenden und dadurch auf die zur Umsetzung des Konzepts virtueller Adressräume notwendige Hardwareunterstützung verzichten kann. Die starke Zunahme an Systemen, die die Vorteile von „safe language environments“ nutzen, leistet der Entwicklung von „language based systems“ Vorschub. Heute sind bereits vielfach Rechnerinstallationen vorzufinden, bei denen der größte Teil der vorhandenen Betriebsmittel zum Betrieb einiger weniger High Level Language VMs aufgewendet wird. In diesen Fällen könnte es auch im Interesse der Verbesserung der Betriebsmittelverwaltung sinnvoll sein, nicht mehrere „kleine“ VMs zu betreiben, sondern nur eine. In Bild 9 wurden diese beiden Alternativen ein-

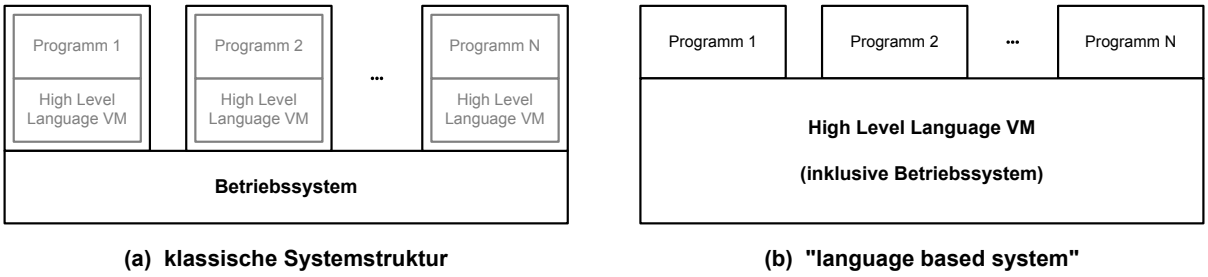


Bild 9: Betriebssystem versus „language based system“

ander gegenübergestellt. Diese Thematik soll hier jedoch nicht weiter vertieft werden, da in Kapitel 4 noch genauer darauf eingegangen wird.

2.2 Java

Die Programmiersprache Java [Gosling et al. 96] ist eine objektorientierte Sprache mit einem strengen Typsystem und C++ ähnlicher Syntax. Java kennt zwei Arten strukturierter Datentypen: Klassen und eindimensionale Arrays. In der Terminologie von Java werden „Zeiger“ auf Objekte und Arrays als Referenzen bezeichnet. Da Java im Hinblick auf die Schaffung eines „safe language environments“ entwickelt wurde, weist das Programmiermodell die hierfür typischen Eigenschaften auf:

- Nur das Kopieren von Referenzen ist erlaubt, jedoch keine „Referenz-Arithmetik“.
- Die Verwendung ungültiger Indizes beim Zugriff auf Arrays wird durch entsprechende Exceptions angezeigt.
- Das Vorhandensein eines „garbage collectors“ wird vorausgesetzt. Die Speicherbereinigung durch den Garbage Collector erfolgt automatisch und muss nicht explizit angestoßen werden.

Das Programmiermodell erlaubt die Formulierung von Nebenläufigkeit durch ein Thread Konzept, das Bestandteil der Sprachspezifikation ist. Neue Threads können während der Programmausführung in beliebiger Zahl erzeugt werden. Für die Synchronisation von Threads bietet Java Unterstützung für die Anwendung des Monitor Konzeptes. Ferner sieht das Programmiermodell die Möglichkeit vor, während der Programmausführung dynamisch weitere Programmteile „nachzuladen“.

2.2.1 Spezifikation der Java VM

2.2.1.1 Ausführung von Java-Bytecode

Die in der Spezifikation der Java VM [Lindholm Yellin 96] beschriebene Funktionsweise der Java VM legt die Assoziation des in Bild 10 dargestellten Grobaufbaus nahe. Links unten ist

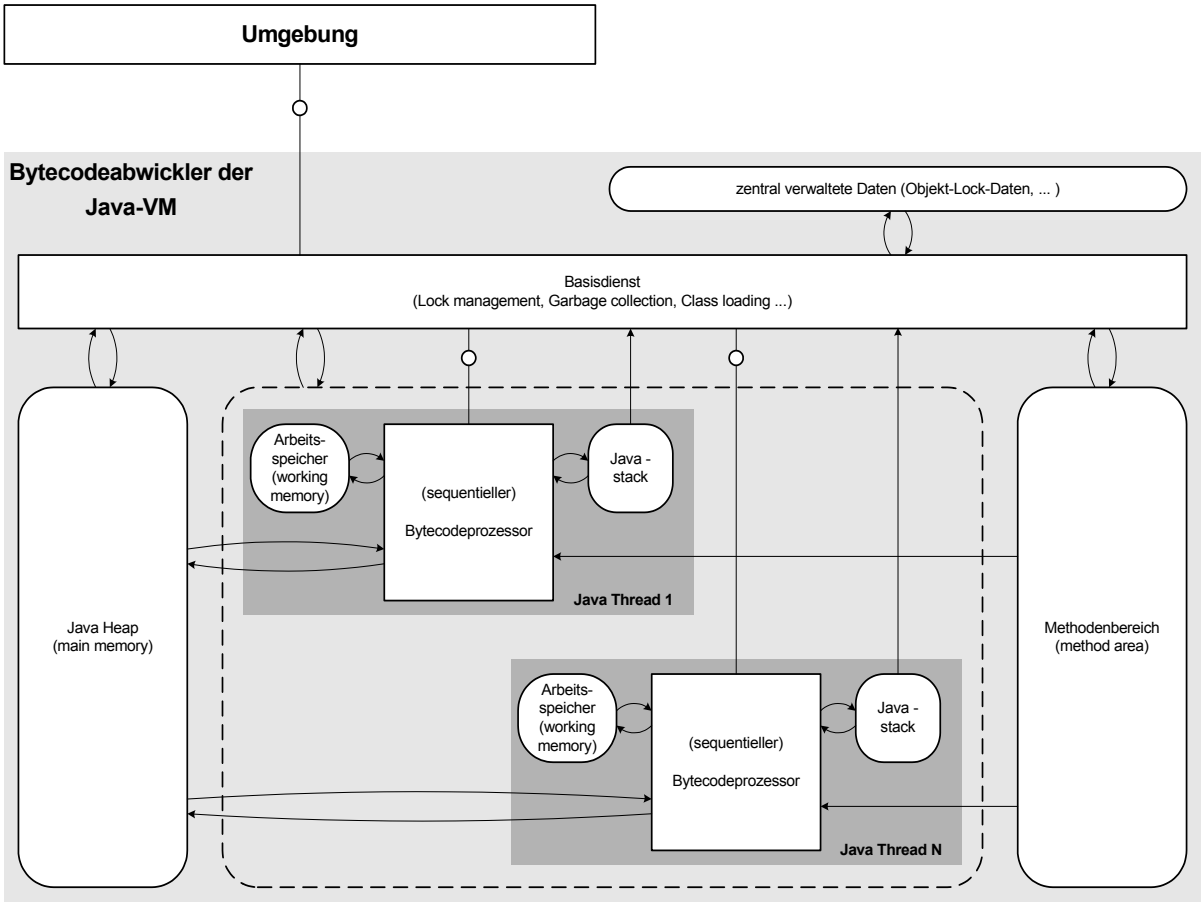


Bild 10: Ausführung von Bytecode durch die Java-VM

der Speicher für den Java Heap dargestellt, der zur Ablage der zu den Java-Objekten gehörenden Daten (Objektattribute) dient. Rechts neben dem Java Heap sind zwei grau hinterlegte Bereiche dargestellt, welche identische Aufbaustrukturen enthalten und auch mit ihrer Umgebung in gleicher Weise verschaltet sind. Im Prinzip hätte diese Struktur nicht genau zweimal, sondern so häufig dargestellt werden müssen, wie es der aktuellen Anzahl an Java-Threads entspricht. Die pro Thread einmal vorhandene Aufbaustruktur besteht aus einem Bytecodeprozessor, einem Stack Speicher für Java Methodenaufrufe sowie einem Arbeitsspeicher. Rechts aussen ist ein als Methodenbereich bezeichneter Speicher dargestellt. Dieser Speicher enthält die bereits geladenen Teile des Java Programms. Er enthält insbesondere die Definition der Methodenimplementierungen in Form von Arrays, die mit Java-Bytecode gefüllt sind. Jeder Bytecodeprozessor ist für die Ausführung aller Bytecode Befehle zuständig, die einem Java Thread zuzurechnen sind und kann auf den Methodenbereich lesend zugreifen. Außerdem hat jeder Bytecodeprozessor Zugriff auf die im Java Heap abgelegten Objektattribute. Bei der Verarbeitung von Objektattributen arbeitet ein Bytecodeprozessor stets auf lokalen Kopien der betreffenden Attribute. Aus diesem Grund ist jedem Bytecodeprozessor ein Arbeitsspeicher zugeordnet, in dem er die Kopien der zu verarbeitenden Objektattribute aus dem Java-Heap ab-

legen kann.[7] Des Weiteren steht jedem Bytecodeprozessor ein als Stack organisierter Speicherbereich für die Realisierung von Methodenaufrufen zur Verfügung. Die in diesem Speicher vorzufindende Datenstruktur ist in Bild 11 veranschaulicht und wird im Anschluss noch genauer vorgestellt. Neben den Bytecodeprozessoren umfasst das Modell der JavaVM noch eine weitere aktive Komponente, die im Bild als Basisdienst bezeichnet ist. Der Basisdienst übernimmt alle Aufgaben, die nicht den Bytecodeprozessoren zugeschrieben werden können. Zu den Aufgaben des Basisdienstes zählt insbesondere das Verwalten von Sperren, das Erzeugen neuer Threads, das Befüllen des Methodenbereichs mit Informationen über die geladenen Klassen und die Garbage-Collection. Insofern erfüllt der Basisdienst im Vergleich zum Bytecodeprozessor eine recht komplizierte Aufgabe. Er wurde in Bild 10 nicht weiter strukturiert, da die Spezifikation den inneren Aufbau dieser Komponente weitgehend offen lässt.

Die in der Spezifikation enthaltenen Aussagen über das Verhalten eines Bytecodeprozessors setzen eine Strukturierung des Java-Stacks voraus. Diese von der Spezifikation vorausgesetzte Strukturierung des Java-Stacks ist in Bild 11 veranschaulicht. Der Java-Stack besteht aus Java-

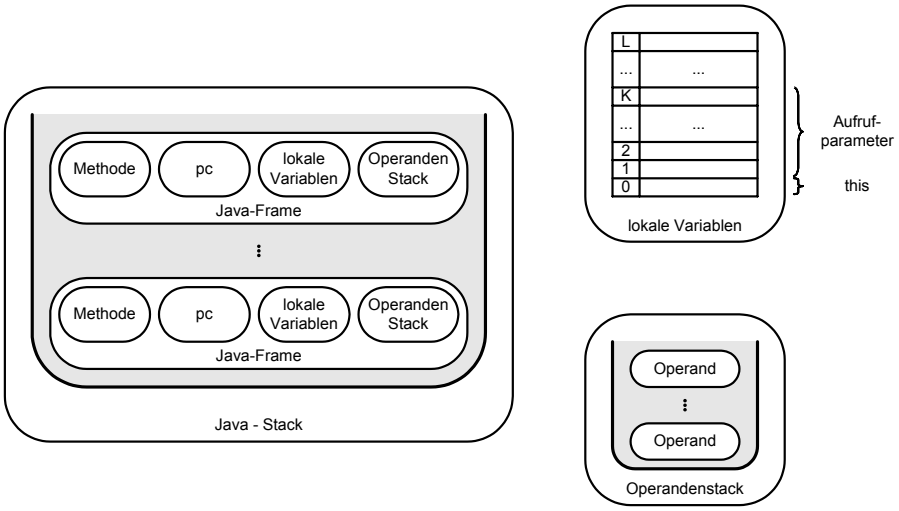


Bild 11: Datenstruktur des Methodenaufrufstacks

Frames. Bei jedem Aufruf einer Java Methode legt der Bytecodeprozessor einen neuen Java-Frame oben auf dem Stack an und entfernt diesen wieder, wenn die Auswertung des Methodenrumpfs abgeschlossen ist. Der oberste auf dem Stack liegende Frame wird auch als aktueller Frame bezeichnet. Ein solcher Frame besteht aus vier Speicherbereichen:

(1) „Methode“
Dieser Speicher enthält einen Verweis auf die im Methodenbereich abgelegte Methodendefinition derjenigen Methode, deren Aufruf mit Hilfe des Frames bearbeitet wird. Die Methode, auf die der oberste Frame des Java-Stacks verweist, wird auch als aktuelle Methode bezeichnet.

(2) „PC“
Der Inhalt dieses Speichers gibt Auskunft darüber, an welcher Position im Bytecodearray der Methode das erste Byte des nächsten auszuführenden Bytecodebefehls zu finden ist.

7. Der Arbeitsspeicher wurde eingeführt, um das mit einem hierarchisch aufgebauten Speichersystem verbundene Caching im Modell der Java VM zu berücksichtigen.

Bei der Erzeugung des Frames wird dieser Speicher mit der Position des ersten auszuführenden Bytecodebefehls der Methode initialisiert (Position 0).

(3) „lokale Variablen“
Dieser Speicher ist selbst wieder in Form eines eindimensionalen Arrays strukturiert. Die Felder dieses Arrays dienen als Ablageorte für die Parameter des Methodenaufrufs, die lokalen Variablen und bei „Instanzmethoden“ auch zur Ablage eines Verweises auf das Objekt bezüglich dessen die Methode aufgerufen wurde („this-Pointer“). Bei einem „normalen“[8] Methodenaufruf wird Element 0 dieses Arrays mit dem „this Pointer“ initialisiert, und die darauffolgenden Array-Elemente werden mit den Werten der aktuellen Aufrufparameter initialisiert.

(4) „Operandenstack“
Hierbei handelt es sich um einen Speicher, der als Stack organisiert ist. Alle Java Bytecodebefehle mit Ausnahme reiner Datentransportbefehle erwarten alle Argumente, die nicht direkt als Teil des Bytecodebefehls angegeben wurden, auf diesem Stack und legen die berechneten Ergebnisse wieder dort ab. Beispielsweise werden bei der Ausführung von Bytecodebefehlen, die der Realisierung der vier Grundrechenarten dienen, jeweils die beiden obersten Elemente von diesem Stack genommen, und das Verknüpfungsergebnis wird wieder oben auf dem Stack abgelegt.

Die erforderliche Anzahl lokaler Variablen sowie die maximale Größe des Operandenstacks werden bei der Übersetzung des Quellcodes für jede Methode getrennt ermittelt und sind Bestandteil der im Methodenbereich abgelegten Methodendefinition.

Das Verhalten eines Bytecodeprozessors kann in erster Näherung als Ausführung der in Bild 12 dargestellten Interpreterschleife beschrieben werden. Die Position eines zu holenden Bytecodebefehls ist durch das Tupel (aktuelle Methode, PC) bestimmt. Der im ersten Byte des Befehls enthaltene opcode identifiziert den Typ der auszuführenden Datenoperation sowie die Art der Bestimmung des nächsten auszuführenden Befehls. Von den 256 möglichen Opcodes sind über

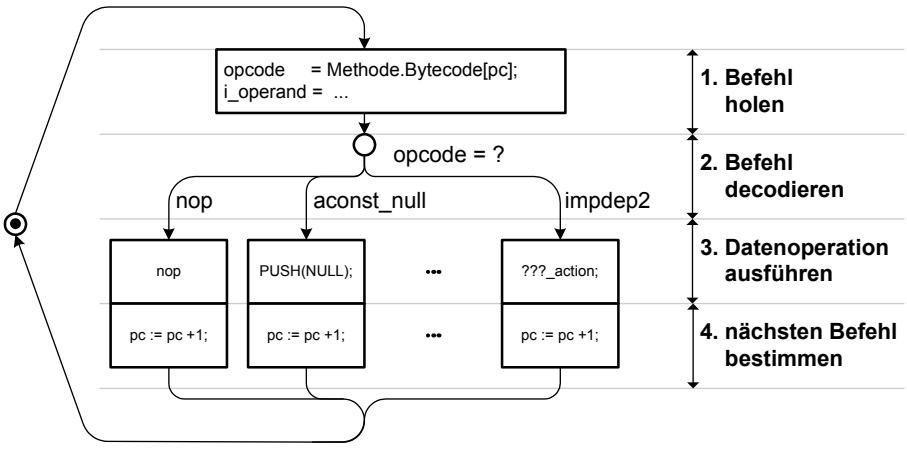


Bild 12: Basiszyklus eines Bytecodeprozessors

200 definiert. Bis auf wenige Ausnahmen kann der Bytecodeprozessor alle Bytecodebefehle

8. Unter einem normalem Methodenaufruf ist hier der Aufruf einer virtuellen Instanzmethode durch den Bytecode mit der mnemonischen Bezeichnung „invokevirtual“ zu verstehen. Die Definition der aufgerufenen Methode muss in Form von Java-Bytecode vorliegen.

ausführen, ohne dass er sich mit dem Basisdienst oder den anderen Bytecodeprozessoren synchronisieren muss. Diese Ausnahmen betreffen folgende Situationen:

- Anforderung und Freigabe von Sperren
  Zur Realisierung des von Java unterstützten Monitor Konzepts ist in der Java-VM eine Sperrverwaltung implementiert. Dabei verwaltet die Java-VM für jedes im Java-Heap abgelegte Objekt eine Sperre.
  Vor dem Freigeben einer Sperre werden die Werte aller im Arbeitsspeicher des Thread befindlichen Arbeitskopien, die seit der Erstellung der Kopie verändert wurden, in den Java-Heap zurückgeschrieben.
  Auch das Anfordern einer Sperre hat Konsequenzen für die aktuell im Arbeitsspeicher des Thread befindlichen Arbeitskopien. Jedoch hat das Anfordern einer Sperre keine Auswirkung auf den Inhalt des Java-Heap, sondern führt lediglich dazu, dass alle im Arbeitsspeicher angelegten Kopien verworfen werden. Der Arbeitsspeicher ist nach der Anforderung einer Sperre leer, und wird bei später erfolgenden Zugriffen auf Objektattribute mit „neuen“ Arbeitskopien befüllt. Darüber hinaus können Veränderungen an Arbeitskopien nach Belieben in den Java-Heap zurückgeschrieben werden.
- Erstmalige Verwendung einer Klasse
  Die Java-VM verfolgt das Konzept des „lazy-loading“. Das heißt, dass die im Methodenbereich abgelegte Information über das Java Programm nicht bereits beim Programmstart vollständig dort zur Verfügung steht, sondern erst bei Bedarf vervollständigt wird. Die kleinste Einheit, um die die im Methodenbereich abgelegte Information ergänzt wird, ist die Definition einer Klasse. Den Bedarf eine Klassendefinition nachzuladen, stellt der Bytecodeprozessor fest, wenn während der Ausführung eines Bytecodebefehls einer der folgenden Fälle eintritt:[9]
    - ein Objekt der betreffenden Klasse erzeugt werden soll
    - erstmalig auf ein „static“ Attribut dieser Klasse zugegriffen wird
    - erstmalig eine „static“ Methode dieser Klasse aufgerufen wird

  Daraufhin unterbricht er die Ausführung dieses Bytecodes und bittet den Basisdienst darum, die fehlende Information im Methodenbereich zur Verfügung zu stellen. Der Basisdienst stellt die Information über die Klasse im Methodenbereich zunächst jedoch nur zur eingeschränkten Nutzung zur Verfügung. Bevor er sie für die „normale“ Benutzung freigibt, müssen noch die in der Klassendefinition enthaltenen „class-initializer“ ausgeführt werden. Die Implementierung dieser „class-initializer“ liegt ebenso wie der Rumpf normaler Java Methoden in Form von Bytecode vor. Die Ausführung dieses Bytecodes gibt der Basisdienst an einen derjenigen Bytecodeprozessoren ab, die ihn um das Nachladen der betreffenden Klassendefinition gebeten haben. Nach erfolgreicher Ausführung der „class-initializer“ teilt der Basisdienst allen Bytecodeprozessoren, die ihn um das Nachladen der betreffenden Klassendefinition gebeten haben, mit, dass die Klassendefinition zur Verfügung steht und dass die Klasse jetzt „normal“ benutzt werden kann. Diese können daraufhin mit der Ausführung des Bytecode-Befehls fortfahren, der zuvor aufgrund fehlender Information unterbrochen werden musste.
- Erzeugen neuer Objekte / Garbage Collection
  Beim Anlegen neuer Objekte muss im Java-Heap Speicher zur Ablage der Attribute des neuen Objekts reserviert werden, diese Speicherallokation muss mit den übrigen Byteco-

9. Die genannten Bedingungen geben die Spezifikation nicht exakt wieder. Die Spezifikation umfasst neben den vorgestellten Bedingungen zahlreiche Ausnahmen und unterscheidet zwischen Interface- und Implementierungsklassen. Ihre Darlegung hätte jedoch in Bezug auf die folgenden Betrachtungen keine neuen Einsichten ermöglicht. Daher wurde hier auf eine exakte Darstellung verzichtet.

deprozessoren und dem Basisdienst synchronisiert werden. Ferner muss der Basisdienst die Garbage Collection des Java-Heap mit den Bytecodeprozessoren synchronisieren. In der Praxis kann man es durch Wahl geeigneter Implementierungstechniken nahezu ausschließen, dass ein Bytecodeprozessor warten muss, weil ein anderer Bytecodeprozessor zeitgleich eine Speicherallokation durchführen will. Im Gegensatz dazu ist es praktisch unmöglich, eine effiziente Speicherverwaltung für den Java-Heap zu implementieren, die gewährleistet, dass die Bytecodeprozessoren während der Garbage-Collection nicht zu längeren Wartepausen gezwungen werden. Die Ursache hierfür sind die Java-Stacks. Die in den Java-Stacks enthaltenen Daten sind einerseits einer hohen Änderungsgeschwindigkeit unterworfen, andererseits enthalten sie Referenzen auf Objekte im Java-Heap, die bei der Garbage-Collection berücksichtigt werden müssen.

- Aufruf von „native“-Methoden
  Die Spezifikation der Java-VM sieht den Fall vor, dass Methoden nicht in Form von Java-Bytecode definiert sind. Der Aufruf einer solchen „native“ Methode führt laut Spezifikation zur Ausführung von „plattformabhängigen Code“. Die Spezifikation der Java-VM [Lindholm Yellin 96] enthält zwar verschiedene Andeutungen, die mit „native“-Code in Zusammenhang stehen, behandelt dieses Thema jedoch nur sehr oberflächlich. Hingegen ist eine Java-VM, die nicht die Möglichkeit bietet „native“-Methoden aufzurufen, unsinnig. Eine solche VM wäre nämlich völlig von der „Außenwelt“ abgeschnitten, da der Basisdienst keine I/O-Funktionalität bereitstellt und auch keine speziellen Bytecode Befehle spezifiziert wurden, die die Kommunikation mit der „Außenwelt“ ermöglichen. Unter Bezug auf Bild 10 ist die Ausführung von „native“ implementierten Methoden eine Aufgabe, die der Umgebung des Bytecodeabwicklers zuzurechnen ist. Da ein Bytecodeprozessor keinen direkten Kanal zur Umgebung besitzt, kann er die Ausführung von „native“ implementierten Methoden nur indirekt über den Basisdienst anstoßen. Der Aufruf von „native“ implementierten Methoden erfolgt stets synchron zur Abwicklung des Java-Bytecodes. Daher darf ein Bytecodeprozessor, der die Ausführung einer „native“ implementierten Methode angestoßen hat, erst dann mit der Auswertung weiterer Bytecode-Befehle fortfahren, wenn die Auswertung des „native“ Methodenaufrufs abgeschlossen ist.

2.2.1.2 Einbettung des Java-Bytecodeabwicklers in seine Umgebung

Die in der Spezifikation der Java-VM enthaltenen Aussagen beschränken sich im Wesentlichen auf ein Ausführungsmodell für Java-Bytecodeanweisungen. Für die in dieser Arbeit betrachteten Systeme ist es jedoch kennzeichnend, dass die Implementierung der Java-VM das Vorhandensein eines Betriebssystems voraussetzt. Die in diesem Fall bestehenden Zusammenhänge zwischen VM-Implementierung und Betriebssystem sollen nun anhand von Bild 13 näher vorgestellt werden.

Der Java-Bytecode-Abwickler, der im letzten Abschnitt beschrieben wurde, ist in diesem Bild in Form des links oben dargestellten Akteurs und des „Java Daten“-Speichers wiederzufinden. Wie im vorangegangenen Abschnitt dargelegt, muss jede Implementierung eines Java-Bytecode Abwicklers über ein „native“-Interface zur Kommunikation mit seiner Umgebung verfügen. Die Umgebung wurde im Vergleich zu Bild 10 weiter strukturiert und besteht nun aus einem „Externen Akteur“, der für die Bearbeitung von „native“ Methodenaufrufen zuständig ist, und aus einem Betriebssystem-Akteur. Um der Tatsache Nachdruck zu verleihen, dass die in Form von „native“-Methoden implementierten Teile des Gesamtsystems Zugriff auf Speicher haben können, der es erlaubt, Daten über die Dauer der Bearbeitung eines „native“ Metho-

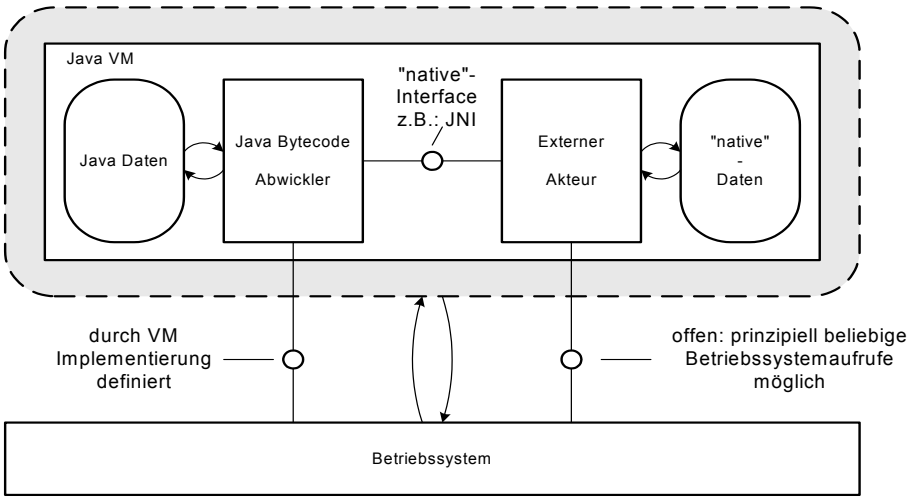


Bild 13: Einbettung des Bytecodeabwicklers in seine Umgebung

denaufrufs hinaus aufzubewahren, wurde auch dem Externen Akteur ein Speicher zugeordnet. Viele Java-VM's unterstützen mehrere „native"-Interfaces, darunter meist das JNI [Liang 99] und ein nicht standardisiertes Interface, das im Hinblick auf die Minimierung des Rechenzeitaufwands bei der Kommunikation zwischen Bytecodeabwickler und Externem Akteur optimiert ist.[10] Bei Implementierungen der Java-VM, die das JNI unterstützen, kann die durch den Externen Akteur bereitgestellte Funktionalität erweitert werden, ohne dass hierzu die Quelltexte der übrigen Teile der Java-VM vorliegen oder neu übersetzt werden müssen. Im Übrigen definiert das JNI eine Schnittstelle, die „in beide Richtungen" benutzt werden kann. Das heißt, dass einerseits der Bytecodeabwickler die Ausführung von „native"-Methoden verlangen kann, andererseits aber auch der Externe Akteur die Ausführung von Java-Methoden durch den Bytecodeabwickler verlangen kann. Sowohl der Java-Bytecode Abwickler als auch der Externe Akteur haben einen Kanal zum Betriebssystem. Während der vom Bytecodeabwickler genutzte Teil der Betriebssystemschnittstelle durch seine Implementierung auf eine bestimmte Teilmenge von Betriebssystemaufrufen eingeschränkt ist, ist die Art der Nutzung der Betriebssystemschnittstelle durch den Externen Akteur zum Zeitpunkt der Entwicklung des Bytecodeabwicklers nicht bekannt. Typischerweise ist eine Java-VM durch einen einzigen Betriebssystemprozesses realisiert. Die spezifikationsgemäße Funktion des Bytecodeabwicklers kann unter diesen Umständen jedoch nur dann garantiert werden, wenn bei der Implementierung des Externen Akteurs bestimmte Regeln bezüglich der Nutzung der Betriebssystemschnittstelle beachtet werden. Diese Regeln werden im weiteren Verlauf dieser Arbeit noch genauer vorgestellt. Der Vertrieb einer Java-VM Implementierung erfolgt normalerweise in Form eines Pakets, das neben der Implementierung des Bytecodeabwicklers auch die Implementierung einer Java-Klassenbibliothek (inklusive passender „native"-Code Teile) beinhaltet.

2.2.2 Beispiel für eine „einfache" Java-VM Implementierung

Bild 14 zeigt ein Aufbaumodell einer vollständigen Java-VM. Die Grobstruktur des Bildes entspricht der von Bild 10. Im Unterschied zu Bild 10 wurde hier jedoch auch der im letzten Abschnitt vorgestellte Aspekt der Ausführung von native Code in die Modellierung mit einbezogen. Das Bild ist in unmittelbarem Zusammenhang mit einer Codeanalyse der CVM-

10. Die vom JNI vorgeschriebene Form der Parameterübergabe beim Aufruf von „native" Code orientiert sich an der „C Calling Convention" der betreffenden Plattform.

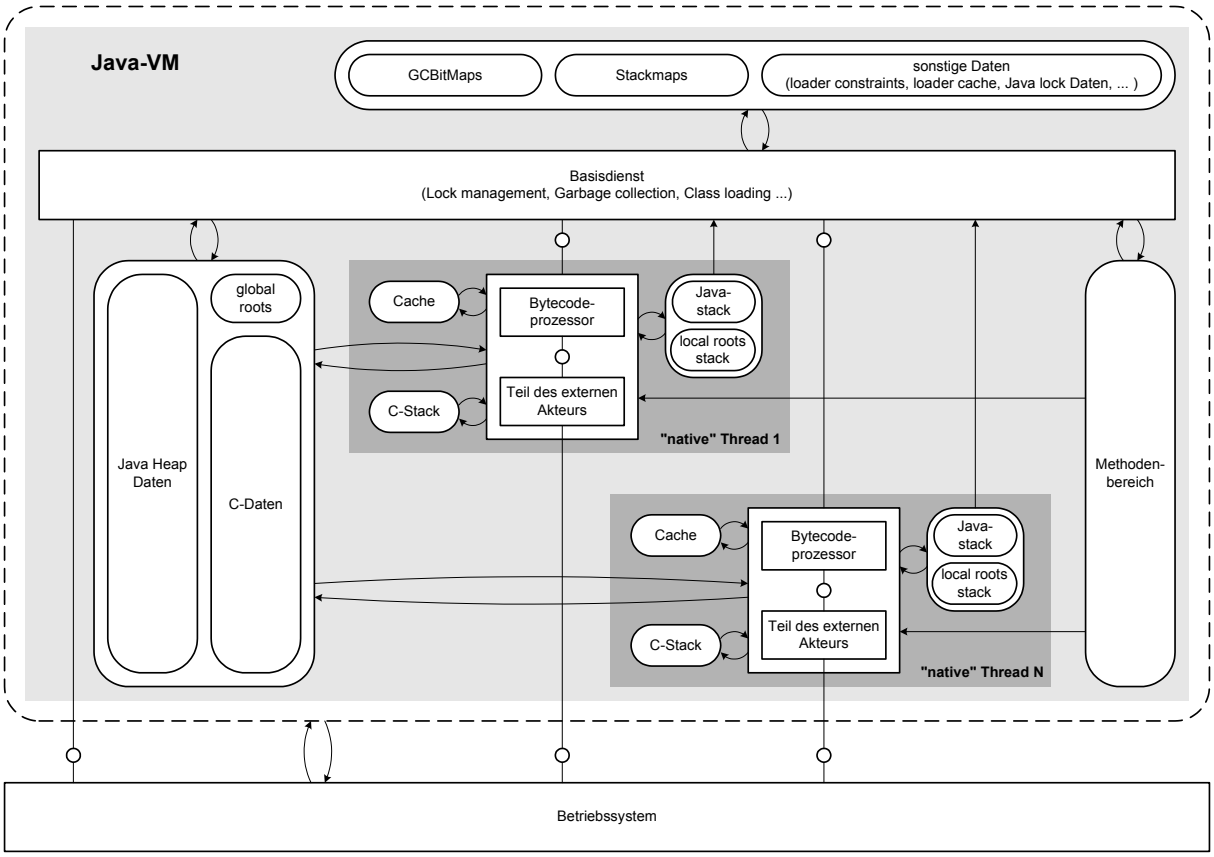


Bild 14: Grobaufbau einer „einfachen“ Java-VM Implementierung

Implementierung der Firma Sun entstanden. Da das Bild sich auf die Grobstruktur der VM beschränkt, sind die im Folgenden anhand des Bildes diskutierten Sachverhalte aber ebenso für das Verständnis anderer Implementierungen von Nutzen.
Die CVM ist in der Programmiersprache „C“ implementiert und setzt ein Betriebssystem voraus, das Multithreading unterstützt. Für jeden Java-Thread wird von der CVM ein korrespondierender „native“-Betriebssystemthread erzeugt. Die Zuordnung zwischen Java-Threads und native Threads ist zeitinvariant, so dass die Auswertung aller Bytecodes eines Java-Threads durch den gleichen native Thread erfolgt. Die Anzahl der von der CVM erzeugten Threads ist im Übrigen stets genau so groß wie die Anzahl der Java-Treads. Die Ausführung von Code, der der Implementierung des Externen Akteurs oder Basisdienstes zuzurechnen ist, erfolgt daher ebenfalls in diesen Threads.
Die CVM-Implementierung umfasst keinen Compiler. Daher ist ein wesentlicher Bestandteil des Programms der native Threads durch die Implementierung der unter Bezug auf Bild 12 vorgestellten Interpreterschleife gegeben.
Da die Realisierung der bei der CVM (standardmäßig) mitgelieferten Java-Class Library ohne eigene Threads auskommt, ist es ausgeschlossen, dass die in Form von native-Code implementierten Teile der Class-Library „selbständig“ die Auswertung von Java-Methoden anstoßen. Stattdessen werden die native Code Teile dieser Library nur infolge des Aufrufs einer „native“ implementierten Class-Library Methode ausgeführt. Es gibt jedoch native implementierte Class-Library Methoden, deren Auswertung selbst wieder Aufrufe von Methoden nach sich zieht, deren Implementierung in Form von Java-Bytecode vorliegt. Trotz dieses Umstandes bleibt die Auswertung des Aufrufs einer „native“ implementierten Class-Library Methode stets ein sequentieller Prozess. Der gesamte Auswertungsprozess einer „native“ implementierten Class-Library Methode erfolgt im Fall der CVM vollständig durch denjenigen native Thread,

der den Bedarf für den Aufruf festgestellt hat. Deshalb wurde der Externe Akteur aus Bild 13 in Form von mehreren gleichen, sequentiell arbeitenden Teilakteuren modelliert und jedem Bytecodeprozessor ein solcher Akteur zur Seite gestellt. Auch der Zustandsspeicher des Externen Akteurs wurde im Unterschied zu Bild 13 zerlegt und ist nun in Form der „C-Stacks“ sowie des von allen Threads genutzten Speichers für „C-Daten“ wiederzufinden.
Bei der Implementierung des Basisdienstes muss besondere Rücksicht darauf genommen werden, dass die an ihn erteilten Aufträge potentiell nebenläufig eintreffen können und dass die Bearbeitung solcher Aufträge im Allgemeinen mit den übrigen Akteuren synchronisiert werden muss. Im Fall der CVM enthält der Code, der der Implementierung des Basisdienstes zuzurechnen ist, daher entsprechende Betriebssystemaufrufe, um die Synchronisation der verschiedenen native Threads während der Ausführung dieses Codes sicherzustellen.
Die Quelltexte der CVM sind so strukturiert, dass die Implementierung des Garbage Collectors leicht durch eine andere Implementierung ersetzt werden kann, die einen anderen Algorithmus zur Garbage Collection verwendet. In den CVM Quellen sind mehrere Implementierungen von Garbage Collectoren enthalten, zwischen denen bei der Übersetzung der Quellen gewählt werden kann. Der vorkonfigurierte Garbage Collection Algorithmus ist ein „Stop The World“ Collector. Das heißt, dass die Ausführung des Java-Programms in jedem Fall für die Dauer der Garbage Collection ausgesetzt wird. Diese Vorgehensweise ist nicht CVM spezifisch, sondern stellt praktisch die Standardvorgehensweise bei der Garbage Collection in Java VM’s dar. Dies liegt einerseits daran, dass die Fortsetzung der Ausführung des Java-Programms während der Garbage Collection erheblichen Zusatzaufwand erfordern würde, um die durch die nebenläufige Ausführung des Java-Programms verursachten Veränderungen am Java-Heap mit der Durchführung der Garbage Collection zu synchronisieren. Andererseits würde allein diese Synchronisation bereits zu einer gravierenden Beeinträchtigung der Ausführungsgeschwindigkeit des Java-Programms führen.
Bei der Durchführung einer Garbage Collection muss der Basisdienst alle Referenzen ermitteln können, die auf Objekte im Java-Heap verweisen. Dies betrifft sowohl Referenzen, die in Form von Objektattributen innerhalb der Java-Heap Daten abgelegt sind, als auch solche, die außerhalb der Java-Heap Daten gespeichert sind. Außerhalb des Java-Heap können Referenzen an den folgenden drei Orten abgelegt sein:

- Java-Stack
- local roots stack
- global roots

Während die Datenstruktur des Java-Stack bereits vorgestellt wurde, tauchen die anderen beiden Speicher in diesem Bild zum ersten Mal auf. Diese Speicher werden von native implementierten Teilen der VM zur Ablage von Referenzen benutzt. Erst dadurch, dass alle Referenzen, die innerhalb der durch native Code implementierten Teile der VM bekannt sind, in diesen Speichern verzeichnet sind, kann der Basisdienst während der Garbage Collection feststellen, welche der im Java-Heap abgelegten Objektdaten verworfen werden können.
In Bild 14 wurde der Speicher für „zentral verwaltete Daten“ des Basisdienstes weiter zerlegt, indem die Speicher für GCBitmaps und Stackmaps von dem Speicher für die sonstigen Daten abgegrenzt wurden. Die Inhalte dieser Speicher stehen in direktem Zusammenhang mit der Garbage Collection.
GCBitmaps nehmen Bezug auf das konkrete Speicherlayout für Objekte einer bestimmten Klasse und geben an, welche der zur Codierung eines solchen Objekts verwendeten Speicherwörter der Codierung von Referenzen dienen.
Entsprechend dienen Stackmaps dazu, Referenzen im Java-Stack zu lokalisieren. Eine Stackmap gibt dabei an, welche der zur Codierung eines Java-Frames verwendeten Speicher-

wörter der Codierung von Referenzen dienen. Welche Stackmap zur Bewertung eines bestimmten Java-Frames verwendet werden muss, hängt einerseits von der Methode ab, deren Aufruf mit Hilfe des Frames bearbeitet wird. Andererseits muss bei der Auswahl der passenden Stackmap auch der aktuelle Bearbeitungszustand dieser Methode berücksichtigt werden. Für die Auswahl der richtigen Stackmap ist es im vorliegenden Fall ausreichend, den aktuellen Wert des PC zum Zeitpunkt der Unterbrechung der Methodenauswertung zu kennen. Dass nicht auch noch weitere Informationen über den aktuellen Bearbeitungszustand des Methodenaufrufs zur Auswahl der richtigen Stackmap erforderlich sind, ist eine besondere Eigenschaft von Java-Bytecode, die auch als „Gosling Property“ bezeichnet wird.

2.2.3 High Performance Realisierungen von High Level Language VM's

In diesem Abschnitt wird auf verschiedene Aspekte der Implementierung von „High Performance“ VMs eingegangen. Obwohl die in diesem Abschnitt vorgestellten Beispiele sämtlich dem Java Umfeld entnommen sind, spiegeln sich die ihnen zugrunde liegenden Konzepte ebenso in Implementierungen anderer High Level Languages wider.

2.2.3.1 N:M Abbildung von Java-Threads auf Betriebssystem Threads

Aufgrund der Tatsache, dass Threads ein fester Bestandteil des Java Programmiermodells sind, ist die Verwendung von Threads in Java-Programmen wesentlich unproblematischer als in einer anderen Programmiersprache wie zum Beispiel „C++“. Außerdem war das Design der Standard Java Klassenbibliotheken lange Zeit so angelegt, dass man bei der Programmformulierung oft nicht um die Verwendung mehrerer Threads umhin kam. Für Systeme, deren Verhalten in der Programmiersprache Java formuliert ist, ist es daher typisch, dass das Systemverhalten als das Zusammenwirken mehrerer (vieler) Threads beschrieben ist. Die CVM ist so konstruiert, dass sie jeden Java-Thread mit Hilfe eines gesonderten Betriebssystemthreads realisiert. Daher konnte bei der CVM auf die Implementierung eines eigenen Schedulers verzichtet werden. Bei modernen Java-VM Implementierungen ist die Abbildung von Java-Threads auf Betriebssystemthreads im Allgemeinen nicht 1:1. Für die Implementierung leistungsfähiger VM's ist es nämlich sehr vorteilhaft, wenn die Scheduling Entscheidungen nicht durch das Betriebssystem des Trägersystems, sondern von der VM selbst getroffen werden. Solche VMs verfügen über einen eigenen Scheduler, der es erlaubt, die meist zahlreich vorkommenden Java Threads mit Hilfe einer wesentlich kleineren Menge von Betriebssystemthreads zu realisieren. Der Scheduler der VM versucht die Anzahl der laufbereiten Betriebssystemthreads dabei stets so groß zu halten wie die Anzahl der physikalisch vorhandenen Prozessoren, die für den Betrieb der VM zur Verfügung stehen. Dadurch wird der Scheduler des Betriebssystems praktisch „ausgeschaltet“. Die Vorteile des Schedulings durch die VM sind vielfältig, die wichtigsten sind:

- Vermeidung von ungünstigen Unterbrechungszeitpunkten
  Zur Ausführung einer Anweisung eines Java-Programms müssen im Allgemeinen mehrere Maschinenbefehle ausgeführt werden. Der Scheduler eines Betriebssystems hat jedoch kein Wissen darüber, wozu die im Rahmen eines Threads auszuführenden Anweisungen dienen und kann dementsprechend auch nicht zwischen günstigen und ungünstigen Unterbrechungszeitpunkten unterscheiden. So ist es im Fall der vorgestellten CVM-Implementierung ungünstig, wenn der einem Java-Thread zugeordnete Betriebssystemthread ausgerechnet dann in den Wartezustand versetzt wird, wenn dieser Thread zuvor Sperren zum Zugriff auf die Datenbereiche des Basisdienstes angefordert hat. Dies kann

leicht zu einer Vervielfachung des Schedulingaufwands führen, weil solche Threads im Allgemeinen andere Threads aufhalten, die ebenfalls auf die Datenbereiche des Basisdienstes zugreifen wollen. Viele solche Wartesituationen können vermieden werden, wenn die VM das Scheduling entsprechend „intelligent“ durchführt.

- Vermeidung von Betriebssystemaufrufen durch Kopplung von Scheduling und Synchronisation
  Wenn, wie im Fall der CVM, jeder Java-Thread durch einen eigenen Betriebssystemthread realisiert ist, kann die Synchronisation der Threads im Allgemeinen nur mit Hilfe von Betriebssystemaufrufen bewerkstelligt werden. Daher zieht diese Realisierungsform üblicherweise einen großen Betriebssystemaufruf-Bedarf nach sich. Der Rechenzeitbedarf der für die Implementierung der Synchronisation von Java-Threads aufgewendet werden muss, kann wesentlich reduziert werden, wenn die VM das Scheduling der Java-Threads übernimmt und der VM-Scheduler mit einer auf ihn abgestimmten Sperrverwaltung gepaart wird.
- Umgehung von Betriebssystembeschränkungen
  Speziell dann, wenn Java-VM’s als Trägersystem für Java-Applikations-Server eingesetzt werden, ist die Anzahl der von der VM zu verwaltenden Java-Threads häufig wesentlich größer als diejenige Anzahl, von der beim Betriebssystementwurf ausgegangen wird.

Problematik „blockierender“ Betriebssystemaufrufe

Wie bereits erwähnt, versucht der Scheduler der Java-VM die Anzahl der laufbereiten Betriebssystemthreads stets so groß zu halten wie die Anzahl der physikalisch vorhandenen Prozessoren, die für den Betrieb der VM zur Verfügung stehen. Bei der Implementierung eines solchen Schedulers stellt die Unterstützung von „native“-Interfaces eine gewisse Herausforderung dar. Dies ist deswegen so problematisch, weil hier berücksichtigt werden muss, dass die Auswertung von „native“ implementierten Methoden auch Betriebssystemaufrufe erforderlich machen kann. Diese Problematik soll nun anhand von zwei alternativen Lösungsstrategien umrissen werden:

(1) direkte Nutzung der Betriebssystemschnittstelle
  Wenn man die VM derart konstruieren will, dass (nahezu) beliebige Betriebssystemaufrufe in „native“ implementierten Methoden vorkommen dürfen, muss man damit rechnen, dass die Ausführung von Threads durch solche Betriebssystemaufrufe „auf unbestimmte Zeit“ ausgesetzt wird. Wenn der Fall eintritt, dass einer der native Threads aufgrund eines Betriebssystemaufrufs blockiert, muss die VM einen zusätzlichen Betriebssystemthread laufbereit machen, so dass wieder für jeden Prozessor ein laufbereiter Thread zur Verfügung steht. Entsprechend muss die VM nach dem Ende einer solchen Blockade, die Anzahl der verwendeten Betriebssystemthreads wieder herabsetzen.

(2) indirekte Nutzung der Betriebssystemschnittstelle
  Eine andere Alternative zum Umgang mit der Problematik von Betriebssystemaufrufen in „native“ implementierten Methoden ist die, dass die VM eine spezielle Schnittstelle zur Nutzung von Betriebssystemdiensten bereitstellt und die direkte Nutzung der Betriebssystemschnittstelle verboten ist. Das mit der Bereitstellung dieser Schnittstelle verfolgte Ziel besteht darin, die von der VM genutzten Betriebssystemthreads stets laufbereit zu halten. Um dieses Ziel zu erreichen, darf die VM keine Betriebssystemaufrufe durchführen, die dazu führen, dass die Ausführung des betreffenden Threads vom Betriebssystem ausgesetzt wird. Dies setzt allerdings voraus, dass die Schnittstelle des Betriebssystems so konstruiert ist, dass dies möglich ist. Diese Vorgehensweise wird im

Übrigen nicht nur in High-Level-Language VM's angewandt, sondern kommt auch in Programmbibliotheken zur Anwendung, die der Realisierung von „user level threads" („green threads") dienen[11].

Beide Lösungsansätze haben Vor- und Nachteile. In realen Java-VM Implementierungen findet man daher oft Mischformen vor. Der zweite Ansatz gibt der VM-Implementierung volle Kontrolle über alle Betriebssystemaufrufe und vereinfacht die Implementierung der VM. Er bringt aber auch gravierende Nachteile mit sich: Einerseits kann bei der Entwicklung von „native" implementierten Methoden nicht mehr auf Standard-Programmbibliotheken zurückgegriffen werden, da zur Realisierung der Bibliotheksfunktionen im Allgemeinen auch Betriebssystemdienste in Anspruch genommen werden. Andererseits können die durch die Standardisierung des JNI geschaffenen Möglichkeiten nicht voll ausgenutzt werden, da der Code von „native" implementierten Methoden dadurch VM spezifisch wird. Durch die Standardisierung des JNI ist es nämlich prinzipiell möglich, die Binaries von „native" implementierten Methoden für unterschiedliche VM Implementierungen zu verwenden[12].

2.2.3.2 Integration von Compilern in virtuelle Maschinen

Heutige Implementierungen von Java-VM's beinhalten üblicherweise einen Compiler, um eine schnellere Programmausführung zu erreichen. Frühe Implementierungen solcher VM's übersetzten jeglichen Bytecode vor der Ausführung zunächst in native-Code. Das Design solcher „Just In Time (JIT)" Compiler war im Hinblick auf geringen Übersetzungsaufwand optimiert. Daher verzichtete man generell auf aufwendige Optimierungen bei der Übersetzung. In der Folge wurde die Programmiersprache Java auch für die Entwicklung von Systemen mit längeren Betriebszeiten eingesetzt. Dadurch wurde es interessant, optimierende Compiler in Java-VM's einzusetzen. Im Allgemeinen ist der Einsatz eines optimierenden Compilers in einer Java-VM jedoch nur dann von Nutzen, wenn der Einsatz des optimierenden Compilers auf diejenigen Teile des Programms beschränkt wird, die später auch entsprechend häufig ausgeführt werden. Java-VM's, die einen Nutzen aus dem Einsatz eines optimierenden Compilers ziehen wollen, müssen daher auch in der Lage sein Bytecode auszuführen, ohne dass dieser zuvor mit dem optimierenden Compiler übersetzt wurde. Außerdem muss die Implementierung einer solchen VM einen Mechanismus zur Abschätzung der Häufigkeit beinhalten, mit der einzelne Programmteile in der Zukunft ausgeführt werden. Frühe Systeme dieser Art beinhalteten daher neben dem optimierenden Compiler noch einen Interpreter, durch den zunächst die Ausführung sämtlicher Programmteile erfolgte. Die Übersetzung eines bestimmten Programmteils wurde üblicherweise initiiert, wenn die Anzahl der Auswertungen eines Programmteils einen bestimmten Grenzwert überschritt. Die Optimierungsentscheidungen solcher Compiler waren üblicherweise konservativ, so dass sie nicht später wieder aufgrund von veränderten Rahmenbedingen revidiert werden mussten. Die in heutigen Java-VM Implementierungen vorzufindende Compilertechnik ist wesentlich ausgefeilter. Um einen Überblick über den Stand der Technik in diesem Bereich zu vermitteln, wird nun kurz auf die Implementierung der „Jikes

11. Ein Beispiel für eine solche Bibliothek die „pth"-Bibliothek [Engelschall 01]. Die von dieser Bibliothek angebotene Schnittstelle zur Nutzung von Betriebssystemdiensten ist syntaktisch weitgehend mit der eines POSIX Systems identisch. Dadurch können bestehende „C"-Programme, die im Hinblick auf ein POSIX-System entwickelt wurden und nur den von der Pth unterstützten Teil der POSIX-Betriebssystemschnittstelle nutzen, praktisch ohne Modifikation der Quelltexte auf „green-threads" umgestellt werden.
12. Die Austauschbarkeit der entsprechenden Binaries ist natürlich nur für solche Java-VM's gegeben, die das gleiche Application Binary Interface voraussetzen (siehe Bild 5).

RVM“[13] [Alpern et al. 00], [Arnold et al. 00-AO] eingegangen. Die Implementierung der Jikes RVM beinhaltet keinen Interpreter, sondern verfügt stattdessen über einen im Hinblick auf geringen Übersetzungsaufwand optimierten Baseline Compiler, der zur erstmaligen Übersetzung des in die VM geladenen Bytecodes verwendet wird. Im Anschluss werden gezielt diejenigen Teile des aktuell in Ausführung befindlichen „native“-Codes durch besser optimierte ersetzt, deren Optimierung gewinnbringend erscheint. Dabei ist es durchaus üblich, dass Programmteile nicht nur einmal, sondern mehrmals neu optimiert werden. Bild 15 zeigt ein Modell der Jikes RVM. Das Bild besteht aus zwei gekoppelten Teilsystemen, die jeweils grau hinterlegt sind.

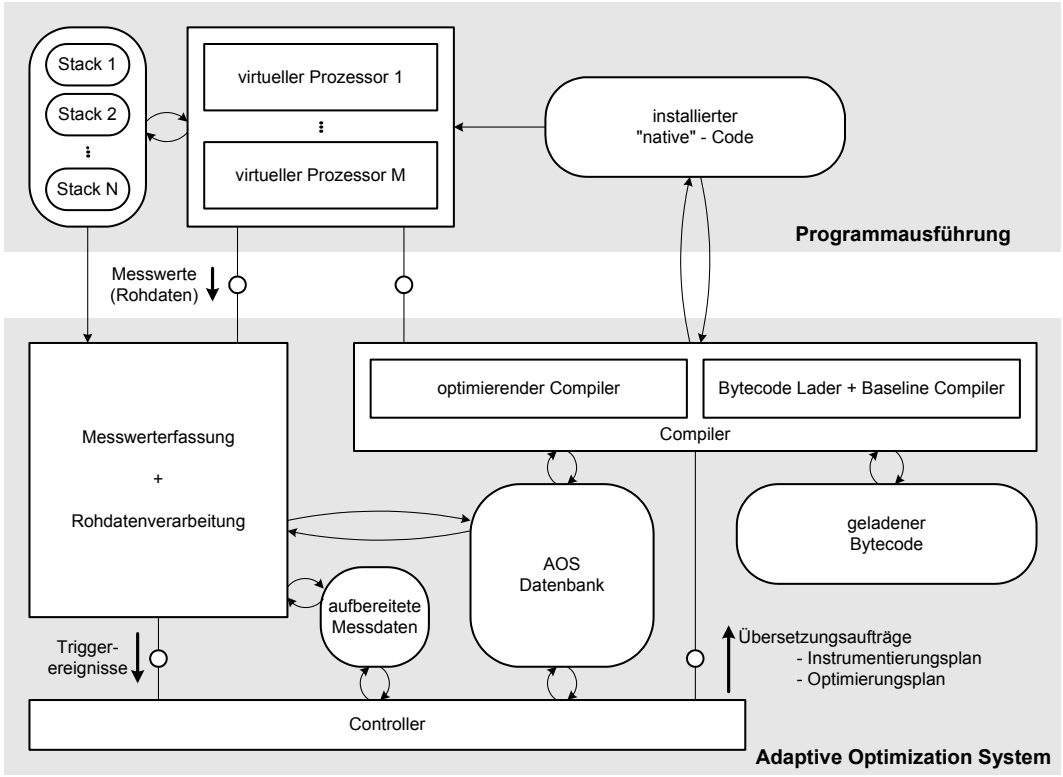


Bild 15: Adaptive Optimierung in der Jikes RVM

Das in der oberen Bildhälfte dargestellte Teilsystem ist für die Ausführung des aktuell installierten native Programmcodes zuständig, während die Aufgabe des unteren Teilsystems darin besteht, den aktuell installierten native Code so zu optimieren, dass die Leistung des Gesamtsystems verbessert wird.
Die Jikes RVM beinhaltet einen eigenen Scheduler, der eine solche N:M Abbildung von Java Threads auf Betriebssystemthreads durchführt, wie sie in Abschnitt 2.2.3.1 beschrieben wurde. In Veröffentlichungen zur Jikes RVM wird davon gesprochen, dass dieser Scheduler die Java-Threads auf „virtuelle Prozessoren“ schedult. Die Anzahl der virtuellen Prozessoren ist üblicherweise der Anzahl der realen Prozessoren gleich. Die Anzahl der verwendeten Stacks ist demgegenüber von der Anzahl der Java-Threads abhängig und daher im Allgemeinen von der Anzahl der virtuellen Prozessoren verschieden.
Das „Adaptive Optimization System“ erfasst während der Programmausführung durch die virtuellen Prozessoren kontinuierlich Messdaten, auf deren Basis im Anschluss die Optimierungsentscheidungen gefällt werden. Für die Erfassung und Aufbereitung solcher Messdaten ist in

13. Die Implementierung der Jikes RVM ist eine Open Source Version der vormals unter dem Namen „Jalapeno JVM“ bekannten Implementierung von IBM und wird insbesondere am IBM T. J. Watson Research Center für Forschungszwecke verwendet.

der Jikes RVM ein eigenes Teilsystem vorgesehen, das im Bild in Form des links außen dargestellten Akteurs wiederzufinden ist. Die eigentliche Entscheidung darüber, ob ein bestimmter Teil des aktuell installierten Codes neu optimiert wird, fällt der Controller. Wenn der Controller den Bedarf für die (erneute) Optimierung eines bestimmten aktuell installierten Codeteils feststellt, beauftragt er den optimierenden Compiler mit der Berechnung des optimierten Codes sowie der anschließenden Installation des neu berechneten Codes. Die Berechnung des optimierten Codes erfolgt dabei nicht auf Grundlage des aktuell installierten Codes, sondern ausgehend vom ursprünglichen Java-Bytecode sowie von Daten, die in der AOS-Datenbank hinterlegt sind.
Zur Messwerterfassung werden verschiedene Verfahren angewandt:

- Instrumentierung des installierten Codes
  Die Compiler reichern den aus der Übersetzung des Bytecodes entstandenen Code mit zusätzlichem Code an, der bei der späteren Ausführung bewirkt, dass die virtuellen Prozessoren die gewünschten Messdaten an die Messwerterfassung übermitteln. Während die Art der Instrumentierung durch den Baseline Compiler unveränderlich ist, kann die Art der Instrumentierung durch den optimierenden Compiler nach Bedarf variiert werden. Der jeweils gewünschte Umfang der Instrumentierung wird dem optimierenden Compiler in Form eines Instrumentierungsplans, der Bestandteil des Übersetzungsauftrags ist, mitgeteilt.
- „Sampling“
  Unter Sampling ist hier eine zyklisch stattfindende Analyse des Inhalts der von den virtuellen Prozessoren benutzten Stacks zu verstehen. Das Bemerkenswerte an der Messwerterfassung durch Sampling ist, dass eine Abtastrate von 10 Millisekunden[14] bereits ausreichend ist, um brauchbare Messdaten zu gewinnen. Zur Messwerterfassung mittels Sampling muss der von den Compilern erzeugte Maschinencode ebenfalls instrumentiert werden, um die regelmäßige Abtastung des Stacks zu triggern. Dass die eigentliche Erfassung der Messdaten in solch großen zeitlichen Abständen erfolgt, hat insbesondere den positiven Effekt, dass das Caching von „Anwendungsdaten“ nicht nachteilig beeinflusst wird.
- „hardware performance monitors“
  Die Erfassung von Performance-Kennzahlen wird heute bereits vielfach durch die Rechnerhardware unterstützt. Im Fall der Jikes RVM gibt es erste Ansätze diese Messdaten auszuwerten. In der Literatur ist jedoch bisher keine High Level Language VM beschrieben, die diese Messdaten für „online“ Optimierungsentscheidungen ausnutzt.

Typische Beispiele für Statistiken, die auf Grundlage dieser Rohdaten erstellt werden, sind:

- „hot methods“
  Unter „hot methods“ sind solche Methoden zu verstehen, für deren Auswertung während der Programmabwicklung überdurchschnittlich viel Rechenzeit aufgewendet wird.
- „calling context tree“ / „receiver class“-Charakteristik
  Der Calling Context tree enthält Informationen darüber, welche Methoden von welchen anderen wie häufig aufgerufen werden.
- „loop unrolling“
  Statistiken über die Anzahl von Schleifendurchläufen

14. Dieser Wert wurde aus [Arnold et al. 00-S1] entnommen. Zur Durchführung der in diesem Aufsatz beschriebenen Experimente wurde ein Rechner des Typs „333MHz IBM RS/6000 PowerPC“ verwendet.

- „branch prediction“
  Statistiken über den Ausgang von Fallunterscheidungen / Häufigkeit des Auftretens von „exceptions“

Alle Akteure des „Adaptive Optimization System“ haben Zugriff auf die AOS-Datenbank. Sie enthält verschiedenste Informationen, von denen hier nur einige erwähnt werden sollen. Der Controller verzeichnet dort beispielsweise, welche Optimierungen bei der Berechnung der verschiedenen aktuell installierten Code-Teile angewendet wurden und wieviel Rechenzeit bei der Durchführung dieser Optimierungen aufgewendet werden musste. Außerdem hinterlegt der Controller dort Richtlinien für die Rohdatenverarbeitung. Über diese Richtlinien kann er einerseits verhindern, dass er zu viele Triggerereignisse erhält, andererseits kann er auch den Aufwand regeln, der zur Messwerterfassung und Rohdatenverarbeitung betrieben wird. Des Weiteren enthält die AOS-Datenbank Analysedaten, die von den Compilern während der Übersetzung von Bytecode erstellt wurden. Zu diesen Analysedaten zählen insbesondere auch Kontrollflussgraphen, welche als Basis zur Annotation von Messdaten verwendet werden.

Wie bereits angedeutet, hat der Controller nicht nur die Möglichkeit zu entscheiden, ob ein bestimmter Teil des installierten Codes optimiert wird oder nicht, sondern er kann differenzierte Optimierungsentscheidungen treffen. Ein an den optimierenden Compiler gerichteter Übersetzungsauftrag umfasst daher detaillierte Angaben bezüglich der anzuwendenden Optimierungen sowie Angaben zum gewünschten Instrumentierungsumfang. Die vom Controller an den Compiler übergebenen Übersetzungsaufträge sind immer derart, dass damit die Neuberechnung des native Codes für eine bestimmte Java Methode verlangt wird. Je nachdem wie häufig bestimmte Codeteile ausgewertet werden müssen, wird der Controller die Übersetzung der betreffenden Methoden mehrfach, mit unterschiedlichen Optimierungs- / Instrumentierungseinstellungen veranlassen.
Nachdem die Berechnung des optimierten Codes erfolgt ist, muss er installiert werden. Da der vom optimierenden Compiler berechnete Code als „Ersatz“ für bereits installierten Code dient, muss die Installation des Codes mit der Ausführung von Code durch die virtuellen Prozessoren synchronisiert werden. Insbesondere kann dabei die Situation eintreten, dass zum Installationszeitpunkt noch Aufrufe derjenigen Java-Methode in Bearbeitung sind, deren Implementierung ersetzt werden soll. Zum Umgang mit solchen Situationen gibt es zwei grundsätzlich verschiedene Verfahrensweisen:

- „On Stack Replacement“
  Eine Möglichkeit zum Umgang mit diesem Problem besteht darin, alle von den virtuellen Prozessoren verwendeten Stacks, in denen Aufrufe der betreffenden Methode enthalten sind, so anzupassen, dass die Auswertung dieser Methodenaufrufe unter Verwendung der neuen Methodenimplementierung fortgesetzt werden kann.
- Gleichzeitige Verwendung verschiedener Methodenimplementierungen
  Die andere Vorgehensweise besteht darin, den neuen Code nur zur Bearbeitung von zukünftigen Aufrufen der betreffenden Methode zu verwenden und Aufrufe, deren Bearbeitung bereits begonnen wurde, vollständig unter Verwendung der alten Implementierung fertig zu bearbeiten.

Beide Verfahren haben Vor- und Nachteile. Der Vorteil des On Stack Replacement besteht darin, dass die durchgeführten Optimierungen sofort nach der Installation des Codes wirksam werden. Im Unterschied hierzu kann die Anwendung des zweiten Verfahrens dazu führen, dass der neu installierte Code erst nach langer Zeit oder überhaupt nie ausgeführt wird. Das Auftreten von langen Verzögerungszeiten ist im Übrigen sogar recht wahrscheinlich, weil es durch eine sehr wichtige Optimierung begünstigt wird, welche als „inlining“ bekannt ist. Die Anwendung

des zweiten Verfahrens hat außerdem Einfluss auf die zulässigen Optimierungen (siehe [Detlefs Agesen 99]), weil die Verwendung von einmal installiertem Code nicht zeitlich begrenzt ist. Die Anwendung von On Stack Replacement ist jedoch nur sehr eingeschränkt möglich. Einerseits können die notwendigen Veränderungen an den Stacks nur mit sehr detailliertem Wissen über die Codierung der in den Stacks vorzufindenden Stack-Frames durchgeführt werden. Allein mit der Bereitstellung dieses Wissens ist im Allgemeinen ein erheblicher Aufwand verbunden. Andererseits ist auch die Durchführung von Veränderungen an den Stack-Frames mit sehr großem Aufwand verbunden. Im Fall der Jikes RVM kommen beide Verfahren zum Einsatz. On Stack Replacement wird jedoch nur im Falle von Methodenaufrufen angewendet, deren Bearbeitung unter Verwendung der vom Baseline Compiler erzeugten Implementierung erfolgt.

2.2.3.3 Compiler-Technik und die Rolle des Austauschformats für Programmteile

In diesem Abschnitt sollen einige wesentliche Unterschiede und Gemeinsamkeiten zwischen normalen „offline“ Compilern und solchen Compilern, die in High-Level-Language VM’s integriert sind, vorgestellt werden.
Bei solchen Compilern, die in VM’s integriert sind, ist die Frage der Minimierung des Übersetzungsaufwandes von größerer Bedeutung als bei „offline“ Compilern, da die Übersetzung während der Programmnutzung stattfindet. Dadurch wird die Anwendung von aufwendigen Optimierungen jedoch keineswegs grundsätzlich ausgeschlossen. Vielmehr sind die in adaptiv optimierenden VM’s vorzufindenden Compiler durchaus in der Lage, fast ebenso aufwendige Optimierungsverfahren anzuwenden, wie „offline“-Compiler. Die Funktionsweise von VM Compilern soll nun anhand des in Bild 16 dargestellten Grobaufbaus des optimierenden Compilers der Jikes RVM diskutiert werden. Der optimierende Compiler der Jikes RVM verwendet

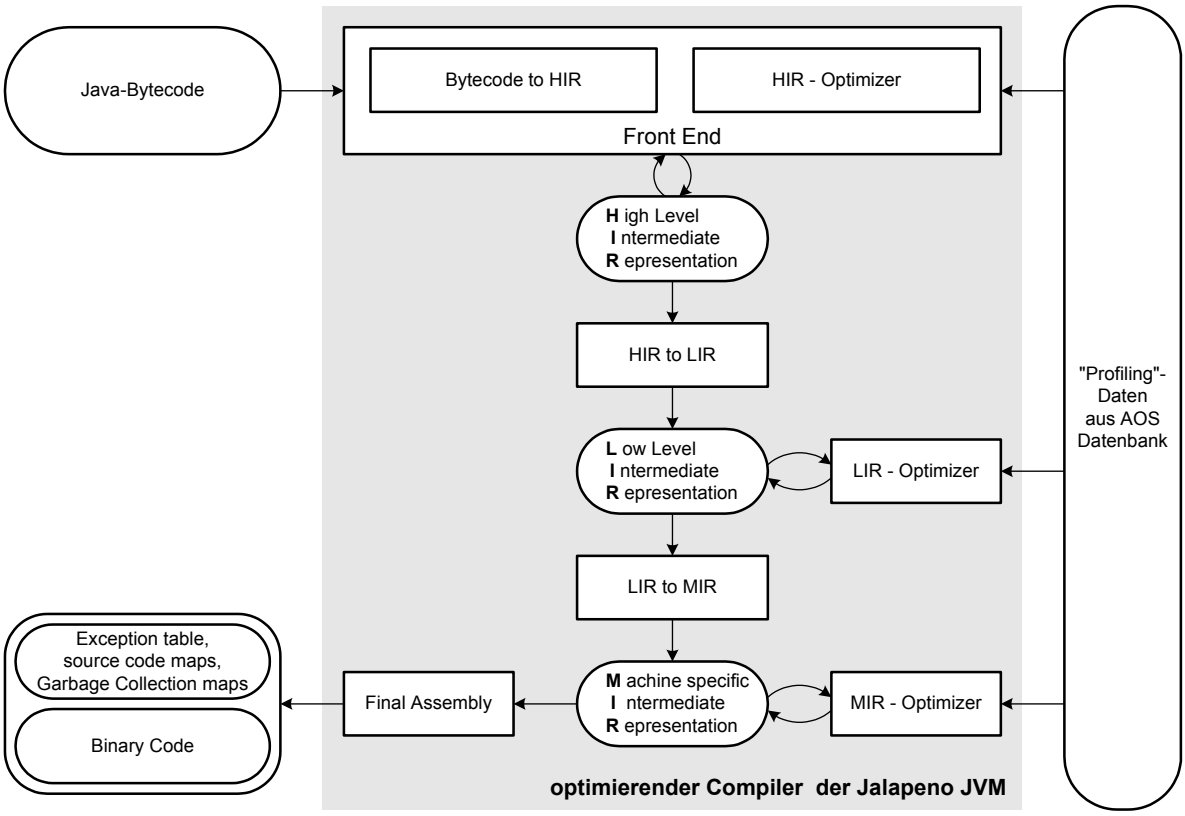


Bild 16: optimierender Compiler der Jikes RVM

intern drei verschiedene Programmdarstellungen. Eine zu übersetzende Methode wird zunächst in eine „High-Level Intermediate Representation (HIR)“ übersetzt, anschließend in eine „Low-Level Intermediate Representation (LIR)“, dann in eine „Machine-specific Intermediate Representation (MIR)“ bevor schließlich das eigentliche Übersetzungsergebnis bestimmt wird. Wie viele „compiler passes“ bei der Übersetzung durch den optimierenden Compiler tatsächlich durchgeführt werden, hängt von dem im Übersetzungsauftrag enthaltenen Optimierungsplan ab. Die in der AOS Datenbank abgelegten Statistiken finden, je nach Optimierungsplan, in allen Optimierungsphasen Berücksichtigung. Das erzeugte Übersetzungsergebnis besteht dabei nicht nur aus Maschinencode, sondern umfasst außerdem noch weitere Informationen, die für die Nutzung des Codes durch die VM notwendig sind. Dazu zählen insbesondere auch die für die Garbage Collection wichtigen Garbage Collection Maps. Garbage Collection Maps nehmen Bezug auf das Speicherlayout eines Stack-Frames, der zur Bearbeitung eines Methodenaufrufs mit Hilfe des erzeugten Maschinencodes dient. Sie geben an, welche der Speicherwörter eines solchen Stack-Frames der Codierung von Referenzen in den Java-Heap dienen.
Allen vom optimierenden Compiler verwendeten „Intermediate Representations“ liegt eine abstrakte Registermaschine zugrunde. Die für die HIR Darstellung benutzten Befehle weisen eine sehr große Ähnlichkeit mit den Java-Bytecode Befehlen auf. Die Unterschiede zwischen den Befehlssätzen sind allein durch die Tatsache begründet, dass der HIR Darstellung eine Registermaschine und der Java-Bytecode Darstellung eine Stack-Maschine zugrunde liegt. Im Unterschied zur HIR Darstellung ist die LIR Darstellung stark von Implementierungsdetails der Jikes RVM geprägt. Dementsprechend werden viele der HIR Befehle bei der Übersetzung in die LIR Darstellung in Sequenzen von einfacheren LIR Befehlen überführt. Im Fall eines HIR Befehls zum Zugriff auf eine Arrayelement, würde der LIR Code beispielsweise eine Befehlssequenz codieren, mit Hilfe derer zunächst eine Überprüfung des Arrayindex durchgeführt wird, anschließend aus dem Arrayindex die Speicheradresse des gewünschten Arrayelements errechnet wird und schließlich der Zugriff auf das Arrayelement erfolgt. Die Datentypen und Befehle des LIR-Codes sind zwar nicht spezifisch für einen bestimmten Rechnertyp, sind jedoch so gewählt, dass möglichst viele „low-level“ Optimierungen bereits unter Bezug auf die LIR Darstellung durchgeführt werden können. Auf diese Weise wird vermieden, dass sie für jeden Rechnertyp, auf den die Jikes RVM portiert werden soll, getrennt implementiert werden müssen. Neben der Einführung der LIR Darstellung wurden weitere Maßnahmen ergriffen, um die Portierbarkeit der Jikes RVM auf andere Rechnertypen zu vereinfachen. Insbesondere wird zur Übersetzung der LIR Darstellung in die für den speziellen Rechnertyp spezifische MIR Darstellung ein BURS-System [Fraser et al. 92] eingesetzt, dessen Implementierung aus solchen für den betreffenden Rechnertyp spezifischen BURS-Regeln generiert wird.

Wie bereits erwähnt, bezieht sich ein an den optimierenden Compiler gerichteter Übersetzungsauftrag immer auf die Bestimmung einer Implementierung für eine ganz bestimmte Methode. Nun ist es für Programme, die in objektorientierten Sprachen abgefasst sind, typisch, dass die Rümpfe von Methoden verhältnismäßig kurz sind. Dementsprechend ist die Anzahl an Bytecodebefehlen, die die Definition einer einzigen Java-Methode darstellen, normalerweise so klein, dass der Code einer einzigen Methode zu wenig Substanz für die Anwendung von Optimierungsverfahren bietet. Voraussetzung für die Anwendung von nachfolgenden Optimierungen ist daher das „Inlining“. Das heißt, dass solche im Rumpf der zu übersetzenden Methode vorkommenden Aufrufe expandiert werden, indem diese Aufrufe durch entsprechend angepasste Versionen der Methodenrümpfe der aufgerufenen Methoden ersetzt werden. Im Fall der Jikes RVM erfolgt dieses Inlining im Rahmen der Erzeugung der HIR Darstellung der zu übersetzenden Methode. Das Problem an dieser überaus wirkungsvollen Optimierung besteht darin, dass der Umfang des erzeugten Codes für eine Methode hierdurch stark zunimmt. Im Fall der Jikes RVM wird das Inlining auf Basis der in der AOS-Datenbank abgelegten Statistiken durchge-

führt und kann daher gezielt auf solche Methodenaufrufe angewendet werden, bei denen der zu erwartende Nutzen hoch ist. Im Gegensatz hierzu stehen „offline“ Compilern keine Statistiken über die Häufigkeit von Methodenaufrufen zur Verfügung. Daher wenden „offline“ Compiler Inlining normalerweise nur in sehr beschränktem Maße an; typischerweise nur dann, wenn der Rumpf der zu „inlinenden“ Methode eine gewisse Maximalgröße nicht überschreitet. Das Inlining ist ein sehr wichtiges Beispiel für eine Optimierung, aus der solche in VM’s integrierte Compiler einen wesentlich größeren Nutzen ziehen können als „offline“ Compiler.

Generell gibt es kaum Optimierungsverfahren, die von solchen in VM’s integrierten Compilern angewendet werden und nicht ebenfalls in „offline“ Compilern Anwendung finden. Vorteile für solche in VM’s integrierte Compiler ergeben sich vielmehr dadurch, dass Optimierungen nutzbringender einsetzt werden können, weil zum Übersetzungszeitpunkt einerseits mehr Information über typische Kontrollflüsse vorliegt und andererseits ein Überblick über sämtliche Programmbestandteile (der aktuell installierte Code) gegeben ist.

Berücksichtigung von „native-Methoden“

Die Anwendung von Optimierungsverfahren setzt im Allgemeinen detailliertes Wissen über den Daten / Kontrollfluss des zu optimierenden Codes voraus. Um solche Methoden, welche Aufrufe von „nativ“-implementierten Methoden enthalten, „optimal“ zu optimieren, wäre es eigentlich erforderlich, den „native Code“ der aufgerufenen „nativ“-implementierten Methoden ebenso detailliert zu analysieren wie den für die Übersetzung relevanten Java Bytecode. Da solche Analysen im Endeffekt ein zweites Compiler-Frontend erforderlich machen würden, wird bei üblichen Java VM Implementierungen auf solche Analysen meist verzichtet.[15] Daher können Methoden, welche Aufrufe von „nativ“-implementierten Methoden enthalten, nur eingeschränkt optimiert werden.

2.2.3.4 Verwaltung des Java Heap / Garbage-Collection

Aufgrund der Fülle an verschiedenen Garbage Collection Algorithmen wäre es unsinnig, an dieser Stelle Details einzelner Garbage Collection Algorithmen vorzustellen. Da es aber je nach Anwendungsprogramm durchaus vorkommen kann, dass der Garbage Collector mehr als die Hälfte der zum Betrieb der VM zur Verfügung stehenden Rechenzeit verbraucht, erscheint es unerlässlich, kurz auf das Thema Garbage Collection einzugehen. Obwohl die Durchführung von Garbage Collection Operationen im Allgemeinen mit großem Aufwand verbunden ist, soll hier explizit darauf hingewiesen werden, dass auch herkömmliche Formen der Speicherverwaltung oft mit ähnlich großem, teilweise sogar größerem Aufwand verbunden sind.

Insgesamt ist das Thema der Bewertung von Garbage Collection Algorithmen recht komplex. Obwohl die Laufzeitkomplexität des Garbage Collection Algorithmus ein sehr wichtiges Bewertungskriterium darstellt, ist es bei weitem nicht das einzige und auch keineswegs immer das wichtigste. So sind bei der Wahl des Algorithmus beispielsweise auch folgende Fragen von Bedeutung: Muss die reguläre Programmausführung für die Dauer der Garbage Collection ausgesetzt werden? Ist es möglich, die Garbage Collection zu parallelisieren? Lässt sich eine Obergrenze für die Dauer einer Garbage Collection garantieren? In welchem Verhältnis steht der Hauptspeicherbedarf zur tatsächlich nutzbaren Heapgröße? Wie aufwendig ist die Objektallokation bei der Verwendung dieses Collectors? Verhindert der Algorithmus die Fragmentie-

15. Ein Prototyp einer Java VM, der auch ein Compiler Frontend für „native Code“ beinhaltet, ist in [Whaley 03] beschrieben.

rung des Heap Speichers? Welchen Einfluss hat er auf das Caching innerhalb des hierarchisch aufgebauten Hauptspeichersystems?
Praktisch alle in modernen Java VMs eingebauten Garbage Collectoren arbeiten inkrementell. Das heißt, dass sie so konstruiert sind, dass sie nicht bei jedem Garbage Collection Vorgang den gesamten Heap untersuchen. Die inkrementelle Garbage Collection setzt jedoch eine Partitionierung des Heap voraus, die es erlaubt, einzelne Teile des Heap getrennt voneinander zu bereinigen. In der Literatur werden eine Reihe von Kriterien zur Aufteilung des Heaps vorgeschlagen, die in unterschiedlichen Kombinationen in heutigen VMs wiederzufinden sind. Hier einige Beispiele solcher Kriterien:

- Aufteilung nach Alter der „Objekte“
  Dies stellt das leistungsfähigste Kriterium dar und wird praktisch von jedem modernen Garbage Collector bei der Aufteilung berücksichtigt. Die Einteilung von Objekten in verschiedene Altersklassen beruht auf der Beobachtung, dass viele „frisch“ erzeugte Objekte sehr bald wieder zu Garbage werden, während solche Objekte, die bereits eine Reihe von Garbage Collections „überlebt“ haben, typischerweise auch noch viele weitere überdauern werden. Daher werden alte Objekte typischerweise in einer separaten Partition des Heaps zusammengefasst, welche wesentlich seltener bereinigt wird.

- Aufteilung nach Erzeugungszeitpunkt / erzeugendem Thread
  Es wurde festgestellt, dass Objekte häufig nicht einzeln, sondern oft in Gruppen erzeugt werden und diese Gruppen auch häufig wieder als Ganzes zu Garbage werden. Wenn diese Gruppen als Ganzes in einer Heap Partition untergebracht werden, kann hierdurch die Anzahl der partitionsübergreifenden Referenzen minimiert werden. Ebenso wurde ein stärkerer Zusammenhalt zwischen Objekten beobachtet, die im Verlauf der Ausführung eines Thread erzeugt wurden.

- Aufteilung nach Objekttyp (im Sinne von Java Klassenzugehörigkeit)
  Diese Aufteilung beruht auf der Beobachtung, dass es häufig einzelne „fruchtbare“ Objekttypen gibt, von denen besonders viele Exemplare erzeugt werden, die jedoch nur eine kurze Lebensdauer haben.

3 Integration von Java/J2EE Technologie in den SAP Application Server

In diesem Kapitel wird die Integration von Java/J2EE Technologie in den SAP Application Server vorgestellt. Damit überhaupt verständlich wird, worin diese Aufgabe besteht, wird im Abschnitt 3.1 zunächst der Begriff des „Application Servers“ allgemein vorgestellt. Danach folgt eine Vorstellung der beiden Application Server Technologien, deren Integration Gegenstand dieses Kapitels ist. Die Technologie des SAP Application Servers wird in Abschnitt 3.2 behandelt, während Abschnitt 3.3 dazu dient, die technologischen Grundlagen von Java basierten Application Servern vorzustellen. Nachdem die technologischen Grundlagen der beiden zu integrierenden Application Server Technologien soweit geklärt wurden, wie es für das Verständnis dieser Arbeit notwendig ist, werden in Abschnitt 3.4 der Bedarf für die Integration der beiden Technologien sowie die damit verfolgten Ziele vorgestellt. In Abschnitt 3.5 wird dann das „VM Container“ Projekt vorgestellt, in dessen Verlauf eine technische Umsetzung der Integration der beiden Application Server Technologien erfolgte, die an dem in Abschnitt 3.4 vorgestellten Bedarf ausgerichtet ist.

3.1 Application Server Technologie

Die SAP hat mit der Konstruktion ihres R/3 Systems erstmalig ein unternehmensweites Informationssystem geschaffen, welches eine so genannte „3-Tier Architektur“ aufweist. Damit unterschied sich ein R/3 grundsätzlich von den damals üblichen Client-/Serversystemen. Diese grundsätzliche Struktur hat sich bis heute erhalten und wurde inzwischen auch von anderen Anbietern solcher Informationssysteme übernommen. Bild 17 zeigt den Grobaufbau eines solchen Systems. Der Rollenverteilung auf die verschiedenen zur Realisierung verwendeten Rechner liegt eine Aufteilung der Systemfunktionalität in drei Schichten zugrunde: Einer Darstellungsschicht, einer Anwendungsschicht und einer Datenbankschicht. Die Datenbankschicht ist durch einen zentralen Datenbankserver realisiert. Er bildet die zentrale Komponente des Systems. Die Verarbeitung der in der Datenbank abgelegten Daten geschieht durch einen Verbund von Application Servern. Jeder Application Server ist dabei für die Bearbeitung von Aufträgen einer bestimmten Menge von Benutzern zuständig. Für jeden durch einen Application Server bedienten Benutzer gibt es auf Seite des betreffenden Application Servers eine spezielle, diesem Benutzer zugeordnete Komponente, die im Bild als „server side user agent“ bezeichnet ist. Die Benutzer haben keinen direkten Zugang zum Application Server, sondern benutzen zum Zugriff auf den Application Server ein auf ihrem Arbeitsplatzrechner installiertes Programm, welches im Bild als „client side user agent“ dargestellt ist. Während der „server side user agent“ für die eigentliche Datenverarbeitung zuständig ist, besteht die Aufgabe des „client side user agent“ primär in der Bereitstellung einer (graphischen) Benutzerschnittstelle. Der „server side user agent“ kann als „Anfragebearbeiter“ charakterisiert werden. Er ist normalerweise passiv und wird erst aktiv, wenn er eine Anfrage vom „client side user agent“ bekommt. Die Bearbeitung der Anfragen seines „client side user agents“ geschieht sequentiell. Bei Erhalt einer Anfrage beginnt er mit der Bearbeitung dieser Anfrage, nach Abschluss der Bearbeitung schickt er das Ergebnis der Anfrage zurück und wartet dann wieder bis er die nächste Anfrage erhält. Der „server side user agent“ verfügt über einen „session data“-Speicher, der es ihm erlaubt, sitzungsbezogene Daten

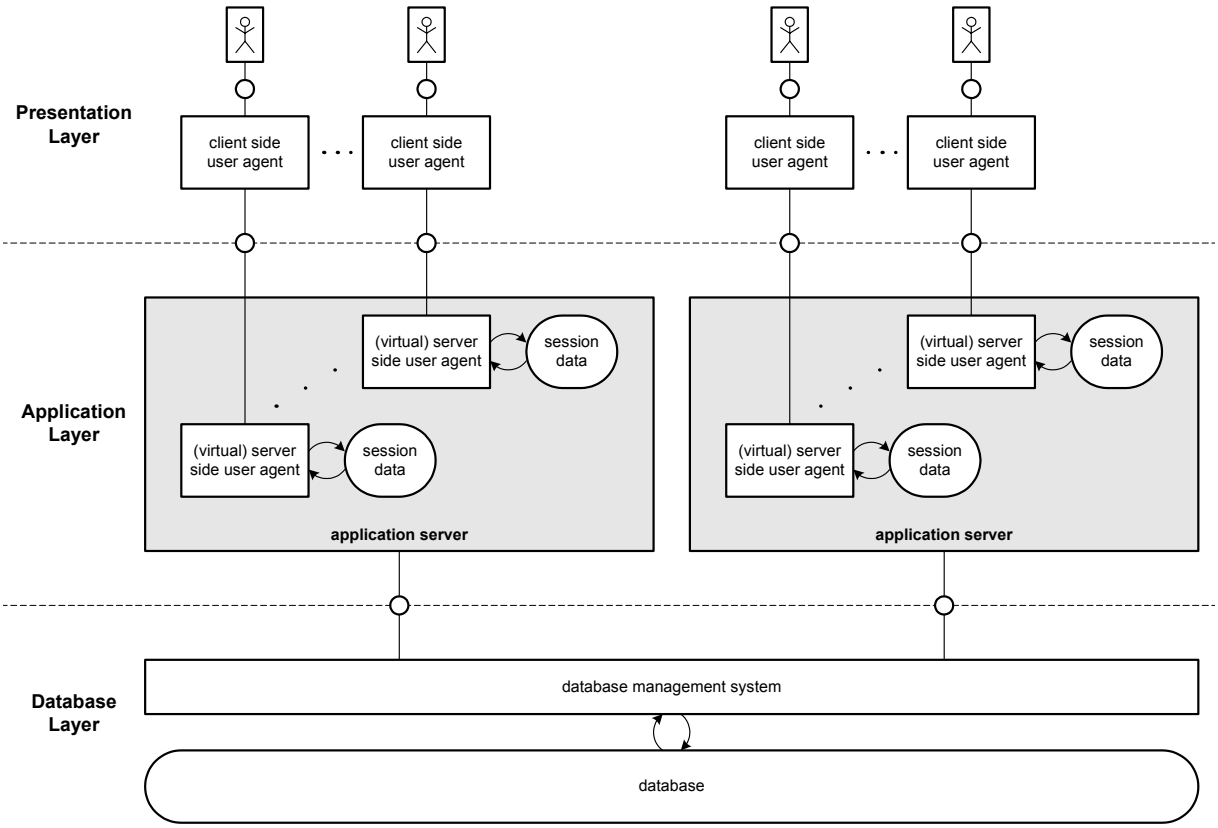


Bild 17: „3-Tier Architektur“

über die Dauer einer Anfrage hinaus aufzubewahren. Verglichen mit dem Zeitraum zwischen zwei Anfragen ist die Dauer der Bearbeitung einer Anfrage typischerweise sehr kurz.

3.2 SAP Application Server Technologie

3.2.1 Dialogprogrammierung

Das Programmiermodell für SAP Anwendungsprogramme basiert auf der Vorstellung, dass die Nutzung eines Anwendungsprogramms als Navigation durch eine Folge von Bildschirmmasken beschrieben werden kann. Die möglichen Maskenfolgen werden dabei vom Anwendungsprogramm vorgeschrieben und können mit Hilfe eines Zustandsgraphen modelliert werden. In der linken Hälfte von Bild 18 ist ein solcher Graph dargestellt. Die Rundknoten eines solchen Graphen sind als Stellvertreter für einen bestimmten Dialogzustand zu interpretieren. Die verschiedenen als Rundknoten dargestellten Dialogzustände äußern sich an der Benutzerschnittstelle in Form unterschiedlicher Bildschirmmasken. Die durch Benutzereingaben auslösbaren Zustandsübergänge sind durch entsprechende Transitionen modelliert. Das Durchlaufen einer bestimmten Maskenfolge entspricht einem Markenfluss durch den Dialoggraphen. Die mit den Nummern 1, 2, 3 versehenen Stellen symbolisieren dabei verschiedene Zustände des Anwendungsprogramms. Die beiden unbeschrifteten Stellen dienen der Modellierung der „Grenzen“ des Anwendungsprogramms. Ist einer dieser Stellen markiert, so bedeutet dies, dass die Nutzung der Anwendung entweder noch nicht begonnen hat oder bereits abgeschlossen ist.

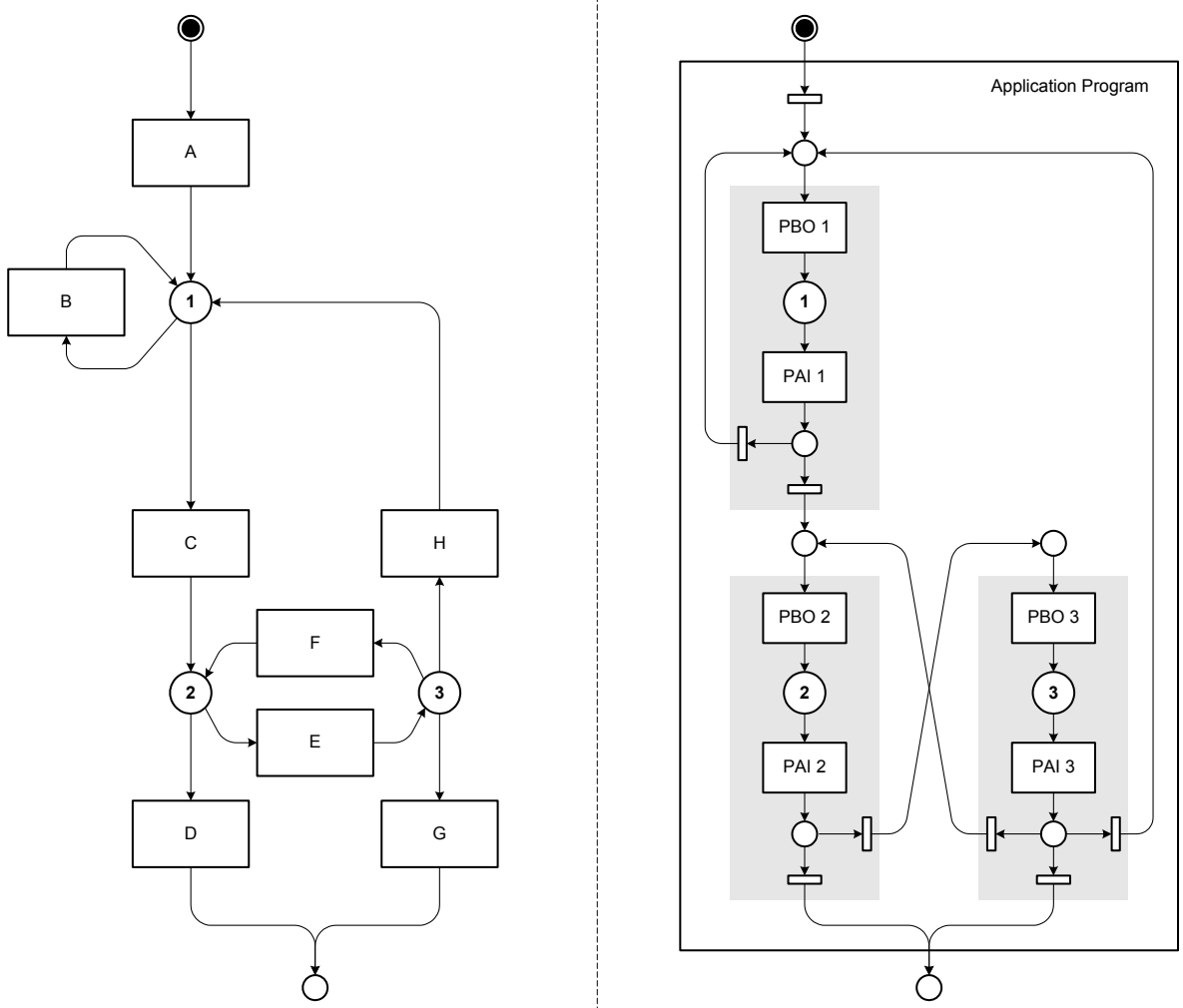


Bild 18: „Navigation Diagram“

Ein Dialoggraph, wie der in der linken Hälfte von Bild 18 gezeigte, erfasst nur Strukturen, die von einem Benutzer der Anwendung am Bildschirm erlebt werden können. Anhand des in der rechten Bildhälfte dargestellten Graphen sollen nun einige für einen Anwendungsentwickler wichtige Begriffe vorgestellt werden. Die beiden dargestellten Graphen repräsentieren zwei verschiedene Sichten auf die gleiche Anwendung. Im Bild äußert sich dieser Zusammenhang zwischen den Graphen darin, dass die im linken Graphen vorkommenden, den unterschiedlichen Dialogzuständen entsprechenden Stellen auch im rechten Graphen wieder vorkommen. Hingegen haben die im linken Graphen vorkommenden, mit Buchstaben beschrifteten Transitionen im rechten Graphen keine direkte Entsprechung. Vielmehr sind im rechten Graphen Transitionen mit der Beschriftung PAI 1 bis 3, PBO 1 bis 3 sowie weitere unbeschriftete Transitionen zu finden. Die Bezeichnung **PBO** steht hierbei für „**p**rocess **b**efore **o**utput“ entsprechend steht **PAI** für „**p**rocess **a**fter **i**nput“. Die mit Nummern versehenen Stellen können nur aufgrund des Schaltens der zugehörigen PBO Transition markiert werden. Ebenso können sie nur durch das Schalten der zugehörigen PAI Transition wieder frei werden. An dieser Stelle muss man bedenken, dass jeder nummerierten Stelle eine bestimmte Bildschirmmaske zuordenbar ist. Dem entsprechend umfasst eine PBO Aktion solche Berechnungen die in unmittelbarem Zusammenhang mit Aufbau der betreffenden Bildschirmmaske stehen, während die Berechnungen die während PAI Aktionen erfolgen der Verarbeitung von Benutzereingaben dienen.

Dem Schalten der im linken Graphen vorkommenden „Dialogschritt“-Transitionen A bis H entspricht dem Schalten von ein oder zwei benannten Transitionen des rechten Graphen. Die grau hinterlegten Bereiche im rechten Graphen deuten eine für Programmierer wichtige Strukturierungseinheit einer SAP-Anwendung an, die als „Dynpro“ (**dyn**amic **pro**gram element) bezeichnet wird. Zur Definition eines Dynpros gehört die Beschreibung einer Bildschirmmaske sowie eine Beschreibung der bei den zugehörigen PBO und PAI Aktionen durchzuführenden Berechnungsschritte[16].

Insgesamt besteht ein klassisches SAP Anwendungsprogramm aus drei verschiedenen Bestandteilen die in Bild 19 dargestellt sind. Sie werden unter Verwendung verschiedener Beschrei-

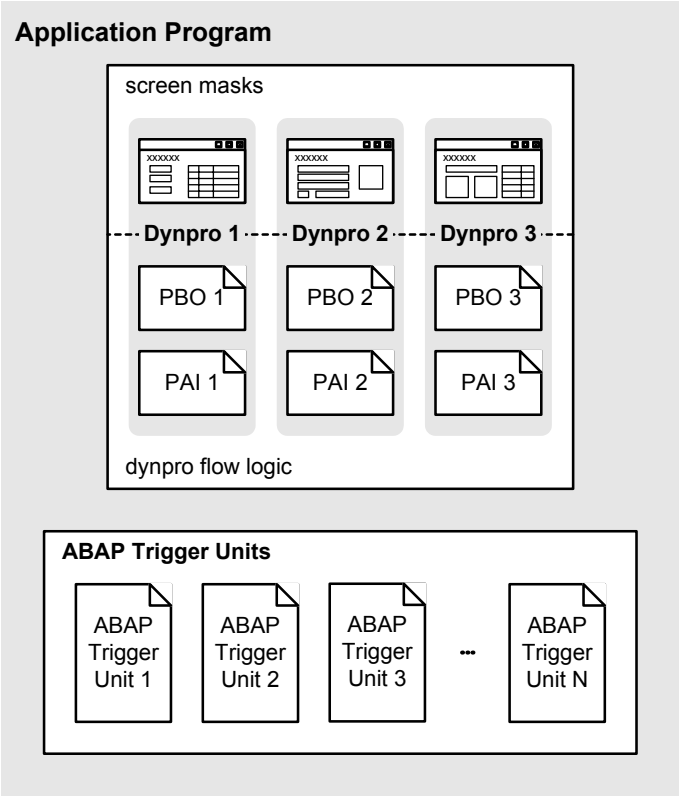


Bild 19: Bestandteile eines klassischen SAP Anwendungsprogramms

bungsmittel beschrieben. Ganz oben sind die zu den Dynpros gehörenden Bildschirmmasken dargestellt, sie werden unter Verwendung eines „screen painters“ definiert. Die übrigen Teile eines Anwendungsprogramms werden textuell, in zwei verschiedenen Programmiersprachen beschrieben. Die Beschreibung der eigentlichen Datenverarbeitungsaufgabe geschieht in der Programmiersprache ABAP („**A**dvanced **B**usiness **A**pplication **P**rogramming“). Die zweite, neben ABAP verwendete Programmiersprache dient zur Formulierung der PBO und PAI Aktionen. Die Beschreibung der PBO und PAI Aktionen wird in der SAP-Terminologie auch als „dynpro flow logic“ bezeichnet. Sie umfasst die Codierung einfacher Plausibilitätsprüfungen für Eingabedaten. Ferner definiert sie, wie „sinnvolle“ Eingabedaten zu verarbeiten sind. Der Code der „dynpro flow logic“ enthält dazu Aufrufe der entsprechenden ABAP Trigger Units. Während die Sprache zur Formulierung der „dynpro flow logic“ in ihren Ausdrucksmöglichkeiten sehr beschränkt ist, ist die Programmiersprache ABAP eine sehr reiche „**4**th **G**eneration **L**anguage“ (4GL-Sprache). Sie verfügt über umfangreiche Ausdrucksmöglichkeiten zum Zu-

16. Die Berechnungsschritte zur Bestimmung des „Folge-Dynpros“ sind Bestandteil der PAI -Aktion.

griff auf Datenbankinhalte, zur Verknüpfung von Daten, zur Formulierung von beliebigen Kontrollflüssen, zur Erzeugung von „Druck“-Dokumenten, ... In jüngerer Zeit wurde sie auch um „objektorientierte“ Sprachelemente erweitert.

Anhand von Bild 20 soll nun der Zusammenhang zwischen den verschiedenen Teilen der Be-

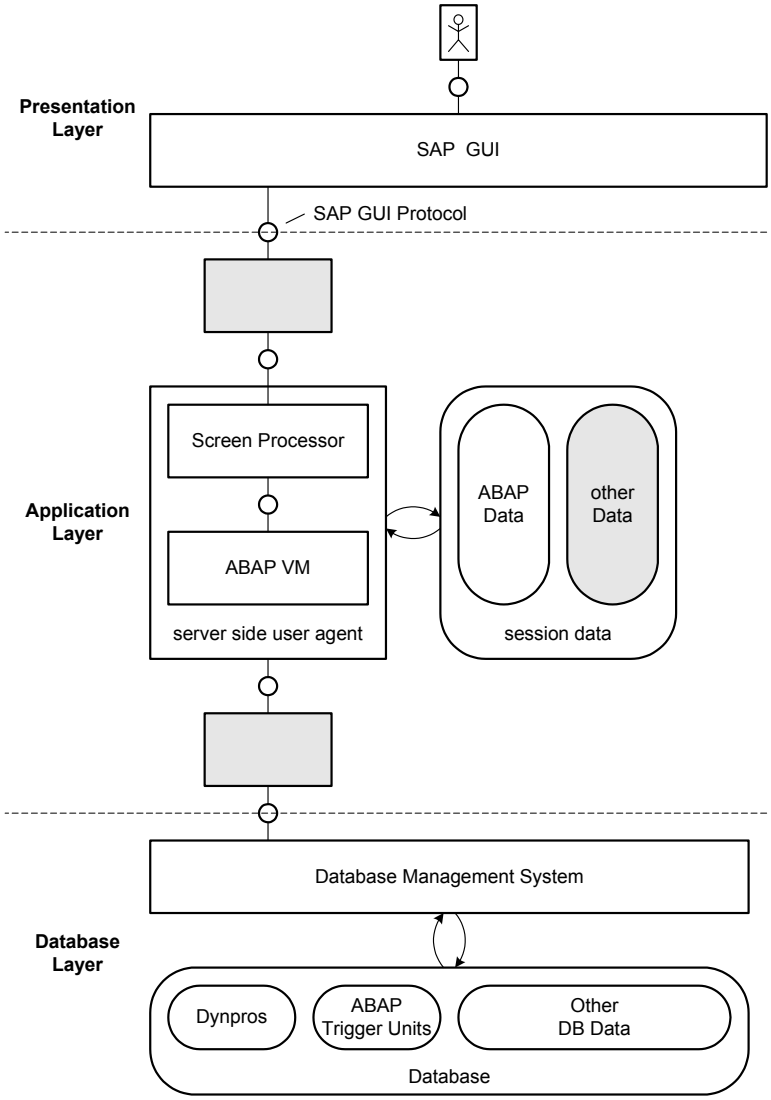


Bild 20: Sicht eines Anwendungsentwicklers auf ein R/3 System

schreibung eines SAP Anwendungsprogramms und der „3-Tier Architektur“ eines SAP-Systems hergestellt werden. Im Fall eines klassischen SAP R/3 Systems werden die „client side user agents“ durch ein Programm realisiert, das als „SAP GUI“ bezeichnet wird. Wenn ein Benutzer sich mit Hilfe des SAP GUI am System anmeldet, nimmt das SAP GUI Kontakt zu einem der vorhandenen Application Server auf. Im Zuge der Anmeldung am Application Server wird ein neuer „server side user agent“ erzeugt und die Verbindung zwischen dem neu erzeugten „server side user agent“ und dem betreffenden SAP GUI hergestellt. Ein „server side user agent“ ist aus Sicht eines Anwendungsprogrammierers mit Hilfe zweier Komponenten realisiert, einem „Screen Processor“ und einer „ABAP VM“. Der dem „server side user agent“ zugeordnete Speicher für die Sitzungsdaten ist ebenfalls in zwei Bereiche unterteilt: Einem „ABAP Data“-Bereich für Daten, auf die explizit im ABAP-Programm Bezug genommen wird, und einem „other Data“-Bereich für Daten, auf die ein Anwendungsprogrammierer keinen direkten Einfluss nehmen kann. Der Speicher für die „other Data“ sowie die beiden unbenannten,

grau hinterlegten Akteure werden später noch genauer vorgestellt. In der zentralen Datenbank eines SAP Systems sind neben den eigentlichen Nutzdaten insbesondere auch sämtliche zur Beschreibung der Anwendungsprogramms gehörenden Programmteile abgelegt.
Die Auswertung der „dynpro flow logic“ erfolgt durch den „Screen Prozessor“ während die Auswertung der in ABAP beschriebenen Funktionalität eines Anwendungsprogramms durch die ABAP VM erfolgt. Damit der „Screen Prozessor“ die Auswertung von „ABAP Trigger Units“ anstoßen kann, besitzt er einen entsprechenden Kanal zur ABAP VM. Die textuellen Beschreibungen der „dynpro flow logic“ und der „ABAP Trigger Units“ werden zunächst in eine Zwischensprache übersetzt, bevor sie vom „Screen Prozessor“ beziehungsweise von der ABAP VM ausgeführt werden können. In beiden Fällen handelt es sich bei der Zwischensprache um eine von der konkreteten Application Server Hardware unabhängige Sprache. Die Übersetzung von Quelltexten in die betreffende Zwischensprache erfolgt erst dann, wenn die darin beschriebene Funktionalität zum ersten Mal genutzt werden soll. Die gewonnenen Übersetzungsergebnisse werden ebenfalls in der zentralen Datenbank abgelegt, so dass sie nicht stets neu erzeugt werden müssen.
An dieser Stelle soll betont werden, dass das Programmiermodell für SAP Anwendungsprogramme explizit so angelegt ist, dass ein Anwendungsprogrammierer die Vorstellung haben darf, dass jedem Benutzer eine eigene ABAP VM und ein eigener Screen Prozessor zur Verfügung steht.

3.2.2 Abbildung von „server side user agents“ auf Betriebssystemprozesse

Als Trägersystem für die „client side user agents“ dienen die Arbeitsplatzrechner der betreffenden Benutzer. Da praktisch jedem Benutzer ein eigener Rechner zur Verfügung steht, hat das Design der „client side user agents“ kaum Einfluss auf die Skalierbarkeit des Gesamtsystems. Im Unterschied hierzu hat die Realisierung der „server side user agents“ wesentlichen Einfluss auf die Skalierbarkeit des Gesamtsystems. Das Design der „server side user agents“ wird wesentlich durch zwei Kriterien bestimmt:
Einerseits ist es wichtig, die Anzahl an „Application Server“-Rechnern klein zu halten. Dies stellt nämlich die Grundvoraussetzung zur Implementierung eines effizienten Cachings von Datenbankinhalten dar und verringert außerdem den Aufwand, der für die Verwaltung des „Application Server Clusters“ betrieben werden muss. Die Minimierung der Anzahl an Application Servern steht in direktem Zusammenhang mit der Minimierung des Betriebsmittelbedarfs, der zur Realisierung der „server side user agents“ benötigt wird.
Andererseits ist es für einen stabilen Betrieb des Systems von größter Bedeutung, die einzelnen „server side user agents“ so gegeneinander abzuschotten, dass ein durch ein fehlerhaftes Anwendungsprogramm bewirkter „crash“ eines „server side user agents“ für die „server side user agents“ anderer Benutzer ohne Konsequenzen bleibt.

Es ist auf den ersten Blick naheliegend, die Abschottung der verschiedenen „server side user agents“ dadurch zu bewerkstelligen, dass man jeden „server side user agent“ durch einen eigenen Betriebssystemprozess realisiert. Im Verlauf der Entwicklung des SAP Application Servers hat sich diese Form der Realisierung jedoch nicht durchsetzen können. Zur Maximierung des Request-Durchsatzes hat es sich insbesondere als zweckmäßig erwiesen, die Anzahl der gleichzeitig in Bearbeitung befindlichen Requests zu begrenzen.

Bild 21 zeigt den Aufbau eines SAP Application Servers. Ein SAP Application Server benutzt zur Realisierung der „server side user agents“ einen „Dispatcher“-Prozess sowie einen Pool von „Work“-Prozessen. Die Vielzahl der durch den Application Server zu realisierenden „server

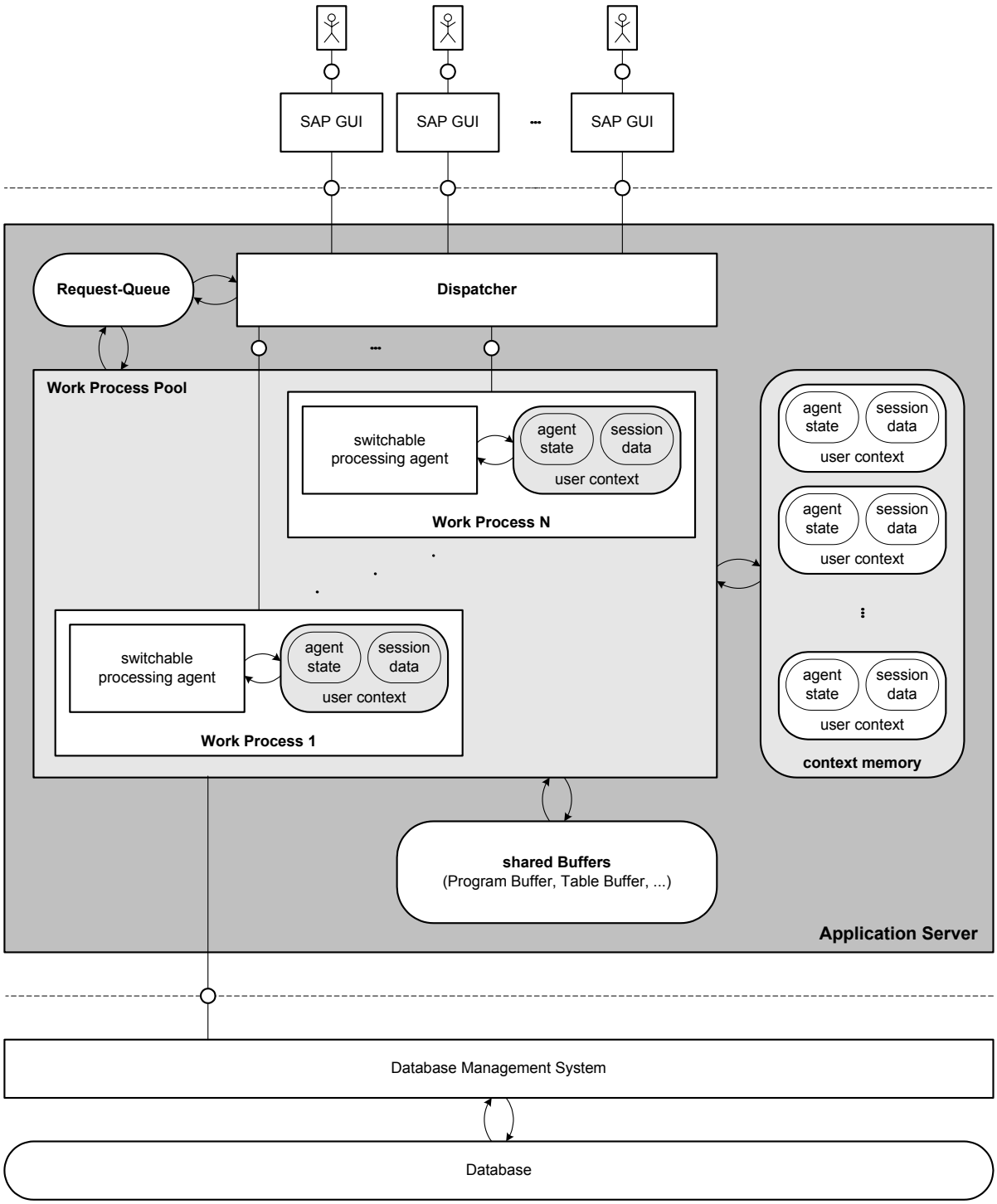


Bild 21: SAP Application Server

side user agents“ äußert sich im Bild in Form einer entsprechenden Anzahl an „user contexten“, welche im „context memory“ abgelegt sind. Für jeden zu realisierenden „server side user agent“ existiert auf dem Application Server ein entsprechender „user context“. Ein „Work“-Prozess kann als umschaltbarer „server side user agent“ betrachtet werden. Die Umschaltung eines „Work“-Prozesses zwischen verschiedenen „server side user agents“ geschieht durch Austausch des „user context“ des betreffenden „Work“-Prozesses. Die „Work“-Prozesse sind „single threaded“. Ihre Anzahl ist von der Anzahl der durch den Application Server realisierten „server side user agents“ unabhängig. Sie orientiert sich vielmehr an der Anzahl der physika-

lisch vorhandenen Prozessoren des Application Servers. Für die Koordination der „Work“-Prozesse ist der Dispatcher zuständig. Er nimmt die von den SAP GUI’s kommenden Requests entgegen, puffert sie in der Request-Queue und teilt sie den „Work“-Prozessen zur Bearbeitung zu. Die im Rahmen der Bearbeitung des Requests von einem „Work“-Prozess erstellten Antwortdaten legt dieser ebenfalls im Request-Queue Speicher ab. Wenn die Bearbeitung eines Requests abgeschlossen ist, meldet der betreffende „Work“-Prozess dies dem Dispatcher, der daraufhin die erzeugten Antwortdaten an das entsprechende SAP GUI versendet. Alle „Work“-Prozesse benutzen einen gemeinsamen, im Bild mit „shared Buffers“ bezeichneten Speicher zur Pufferung von Datenbankinhalten. Die drei im Bild vorkommenden Speicher für die „Request Queue“, das „context memory“ und die „shared Buffers“ sind in Form von „shared memory“ realisiert.[17] Durch diese Form der Realisierung wird die Erstellung von Kopien beim prozessübergreifenden Datenaustausch praktisch vollständig vermieden. Dies ist insbesondere für die Implementierung des „user context“ Austauschs von entscheidender Bedeutung und führt letztlich dazu, dass Aufwand für einen „user context“ Austausch nahezu vernachlässigbar wird. Hierzu ist, wie gleich noch genauer erklärt wird, lediglich eine der Anpassung der Speicherschutzeinstellungen des betreffenden „Work“-Pozesses notwendig. Nun bringt die Verwendung von „shared memory“ grundsätzlich die Gefahr mit sich, dass die beabsichtigte, wechselseitige Abschottung der verschiedenen „Work“-Prozesse untergraben wird. Daher wird im Fall der „Work“-Prozesse jeweils nur der „user context“ in den Adressraum eines „Work“-Prozesses eingeblendet, der für die Bearbeitung des aktuellen Requests benötigt wird. Der für die Request-Queue verwendete Speicherbereich wird, mit Ausnahme des Bereichs in dem die im Rahmen der Bearbeitung des aktuellen Requests erstellten Antwortdaten abgelegt werden, nur „read-only“ in den Adressraum des betreffenden „Work“-Prozesses eingeblendet. Der Speicherbereich für die „shared Buffers“ ist üblicherweise ebenfalls nur read-only in den Adressraum eingeblendet. Zur Durchführung von Veränderungen am Pufferinhalt werden die Speicherschutzeinstellungen vorübergehend gelockert. Da die „Work“-Prozesse „single threaded“ sind, ist jedoch sichergestellt, dass während dieser Zeit nur ganz bestimmte Codeteile ausgeführt werden.
Durch diese Art der Realisierung wird einerseits eine effiziente Umschaltung der „Work“-Prozesse zwischen verschiedenen „server side user agents“ ermöglicht, andererseits wird verhindert, dass „außer Kontrolle“ geratene „Work“-Prozesse neben dem aktuell in den Adressraum eingeblendeten „user context“ noch weitere „user contexte“ unbrauchbar machen können.

An dieser Stelle soll noch einmal näher auf die Frage eingegangen werden, warum nicht jeder „server side user agent“ durch einen eigenen Betriebssystemprozess realisiert wird. Schließlich wäre diese Form der Realisierung einfacher und würde ebenfalls die Verwendung eines „shared Buffers“ ermöglichen, der auf Grundlage von „shared memory“ realisiert ist. Im Folgenden wird diese alternative Form der Realisierung, bei der jeder „server side user agent“ durch einen speziell zu diesem Zweck erzeugten Betriebssystemprozess realisiert wird, als Userprozess Konzept[18] bezeichnet. Das Workprozess Konzept ermöglicht im Vergleich zum Userprozess Konzept eine effizientere Nutzung der vorhandenen Betriebsmittel. Dies betrifft sowohl den Bedarf an Hauptspeicher der zur Realisierung eines „server side user agents“ notwendig ist, als auch den Bedarf an Rechenzeit zur Bearbeitung eines Requests. Außerdem erlaubt das Workprozess Konzept die Umgehung üblicher Betriebssystembeschränkungen[19], die im Fall des

17. Die genannten „shared memory“ Bereiche sind durch „memory mapped files“ implementiert. Sie werden von jedem Prozess, der den betreffenden „shared memory“ Bereich nutzt, an die gleiche Stelle im virtuellen Adressraum eingeblendet, so dass die darin abgelegten Zeiger (Pointer) ihre Gültigkeit behalten.
18. Diese Form der Realisierung ist keineswegs ungewöhnlich und wird in der Literatur auch gelegentlich als „forking server“ bezeichnet.

Userprozess Konzepts möglicherweise beschränkend wirken können. Ein wesentlicher Faktor für die effizientere Nutzung der vorhandenen Betriebsmittel liegt in der Begrenzung der gleichzeitig in Bearbeitung befindlichen Requests. Die Anzahl der gleichzeitig in Bearbeitung befindlichen Requests ist hier nämlich auf die Anzahl der benutzten Workprozesse beschränkt. Die Workprozesse sind im Übrigen „single threaded“ und ihre Anzahl ist prinzipiell so gewählt, dass für jeden laufbereiten Workprozess auch stets ein Prozessor zur Verfügung steht. Durch diese Begrenzung der Anzahl der nebenläufig bearbeiteten Requests und die daraus resultierende „batch“-artige Bearbeitung von Requests ergeben sich viele Vorteile:

(1) „direkte“ Vorteile
Dadurch, dass die Anzahl der Workprozesse idealerweise so gewählt ist, dass die Anzahl laufbereiter Threads stets genau so groß ist wie die Anzahl der verfügbaren Prozessoren, wird der Scheduler des Betriebssystem praktisch „ausgeschaltet“. Im Vergleich zum Userprozess Konzept, wo jeder „server side user agent“ durch einen eigenen Prozess realisiert ist, werden bei der Nutzung des Workprozess Konzepts solche Prozesswechsel vermieden, die sonst zur Einhaltung der „fairness“ zwischen allen Betriebssystemprozessen durchgeführt würden. Der Vorteil der sich aus der Einsparung dieser Prozesswechsel ergibt, erwächst dabei im Wesentlichen aus der verbesserten Cache Nutzung und nicht aus dem eingesparten Aufwand für den Austausch des Prozessorkontextes.
Speziell beim Zugriff auf die zentrale Datenbank kann es vorkommen, dass die Bearbeitung eines Requests durch einen Workprozess ausgesetzt werden muss, bis die betreffende Datenbankoperation durchgeführt ist. Durch das Workprozess Konzept wird dabei sichergestellt, dass nach der Beendigung einer solchen Wartesituationen, sofort wieder ein Prozessor für die weitere Bearbeitung des betreffenden Requests zur Verfügung steht.
Dadurch, dass die Bearbeitung von Requests, deren Bearbeitung einmal begonnen wurde, zügig zu Ende geführt wird, ergibt sich der Vorteil, dass Betriebsmittel, die speziell zur Bearbeitung eines Requests angefordert wurden, vergleichsweise schnell wieder frei gegeben werden.

(2) „indirekte“ Vorteile
Zu den Betriebsmitteln, die speziell zur Bearbeitung eines Requests angefordert werden können, zählen insbesondere auch zentrale (Datenbank-)Sperren. Die durch das Workprozess Konzept erreichte Verkürzung des Requestbearbeitungsdauer führt insbesondere auch zu einer Verkürzung solcher Sperrzeiten. Dies hat wiederum zur Folge, dass viele der Wartesituationen, die bei Anwendung des Userprozess Konzepts entstehen würden, gänzlich vermieden werden.

(3) verbessertes (Über-)Lastverhalten
Ein weiterer wesentlicher Vorteil des Workprozess Konzepts, liegt in der Verbesserung des (Über-)Lastverhaltens. Bei Verwendung des Userprozess Konzepts würde die Anzahl der gleichzeitig in Bearbeitung befindlichen Requests mit dem Requestaufkommen ansteigen. Durch die parallele Bearbeitung dieser vielen Requests würde der Wettbewerb um die Betriebsmittel des Application Servers sowie um zentrale Sperren verschärft. Die Folge wäre das verstärkte Auftreten von Wartesituationen bei der Bearbeitung von Requests und ein sinkender Requestdurchsatz. Im Gegensatz hierzu, ist ein Absinken des Requestdurchsatzes bei Verwendung des Workprozess Konzepts nicht zu erwarten, da die Anzahl der gleichzeitig in Bearbeitung befindlichen Requests von der Anzahl der zur Bearbeitung anstehenden Requests unabhängig ist und damit kein verschärfter Wettbewerb um die vorhandenen Betriebsmittel entsteht.

19. Eine solche Beschränkung betrifft üblicherweise die maximale Anzahl von Betriebssystemprozessen.

3.3 Java Application Server Technologie

Im Fall des SAP Application Servers geschieht die Entwicklung von Anwendungen unter Verwendung speziell zu diesem Zweck entwickelter Werkzeuge. Insbesondere wird zur Formulierung der so genannten „Business-Logik“ die im Hinblick auf diesen Zweck entworfene Programmiersprache ABAP verwendet. Die Entwicklung von Java Application Servern ist von der Idee geprägt Application Server zu schaffen, die als Trägersystem die Java-VM nutzen. Es ist für solche Systeme kennzeichnend, dass sämtliche Systemteile in der Programmiersprache Java formuliert sind. Das betrifft sowohl diejenigen Systemteile, die der Realisierung des eigentlichen Application Servers zuzurechnen sind, wie auch die „Business-Logik“ der auf dem Application Server zur Ausführung kommenden Anwendungsprogramme. Die verschiedenen Hersteller solcher Java Application Server haben sich inzwischen auf einen Standard geeinigt, der den Namen „**J**ava **2 E**nterprise **E**dition (J2EE)“ trägt. Der Standard zielt insbesondere darauf ab, das Programmiermodell für Anwendungsprogramme soweit zu standardisieren, dass es möglich wird, Anwendungsprogramme zu entwickeln, die auf allen J2EE konformen Application Servern verwendet werden können. Aus diesem Grund befasst sich der Standard mit zwei Themen besonders ausführlich: Das erste Thema ist die Auslieferung und Installation von Anwendungen. Das zweite Thema widmet sich der Frage, welche Systemfunktionalität ein Anwendungsentwickler bei einem J2EE konformen Application Server voraussetzen darf und wie er auf diese Funktionalität bei der Formulierung der „Business-Logik“ zurückgreifen kann. Auf dieses zweite Thema soll nun etwas genauer eingegangen werden, da es die Entwurfsentscheidungen bei der Integration der Java Application Server Technologie und der bestehenden SAP Application Server Technologie wesentlich beeinflusst hat.

3.3.1 J2EE Anwendungsprogrammierung

Die J2EE Spezifikation definiert eine Reihe von Systemteilen und Schnittstellen, die für die Entwicklung von unternehmensweiten Informationssystemen benutzt werden können. Die verschiedenen von der J2EE Spezifikation erfassten „Einzelteile“ können dabei auf unterschiedliche Weise kombiniert werden. Obwohl die J2EE Spezifikation verschiedene Kombinationsmöglichkeiten zulässt, gibt es ein in der Praxis häufig anzutreffendes Anwendungsszenario, welches in Bild 22 dargestellt ist und als Ausgangspunkt für die weitere Vorstellung der J2EE Technologie dienen soll.
Es ist für Systeme, die auf Basis der J2EE Technologie realisiert sind, typisch, dass der wesentliche Teil der Nutzer solcher Systeme mit Hilfe eines „normalen“ Web-Browsers auf das System zugreift. Dementsprechend verwenden die am oberen Bildrand dargestellten Benutzer einen Web-Browser zum Zugriff auf den J2EE Server. Damit Web-Browser überhaupt als „client side user agents“ verwendet werden können, fungiert ein J2EE Server unter anderem als Web-Server und stellt eine von außen erreichbare HTTP-Schnittstelle bereit.
Um die Skalierbarkeit der unter Verwendung der J2EE Technologie entwickelten Systeme zu fördern, sieht die Spezifikation die Möglichkeit vor, die von einer J2EE Anwendung verursachte Rechenlast auf unterschiedliche Rechner zu verteilen. Daher besteht ein J2EE Server im Allgemeinen aus einem Verbund von Rechnern, auf denen so genannte „Web Container“ und „EJB Container“ betrieben werden. Aus Benutzersicht sind diejenigen Rechner, die zum Betrieb der „Web Container“ eingesetzt werden, den zum Betrieb der „EJB Container“ verwendeten Rechnern vorgelagert.
Der in Bild 22 dargestellte J2EE Server besteht aus genau einem „Web Container“ und einem „EJB Container“. Ob die beiden Container tatsächlich unter Verwendung unterschiedlicher

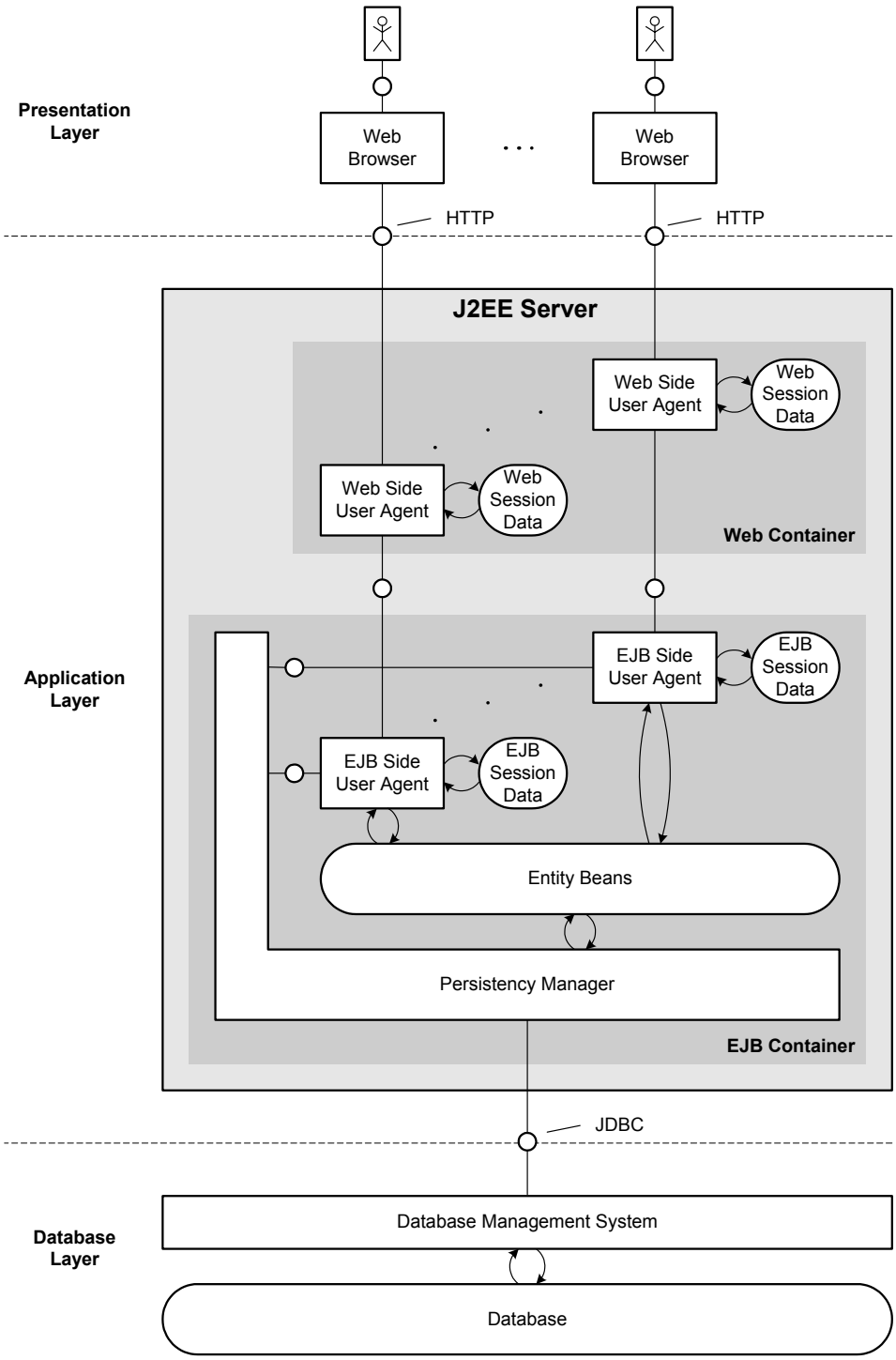


Bild 22: J2EE - Anwendungsszenario

Rechner realisiert sind, ist für die weiteren Betrachtungen unerheblich. Hingegen ist es für das Verständnis wichtig, dass es sich bei den beiden Containern um zwei entkoppelte Teilsysteme handelt, die auch unabhängig voneinander eingesetzt werden können und möglicherweise auf unterschiedlichen Rechnern betrieben werden. Beide Container unterstützen die Bearbeitung von Anfragen, die Benutzersitzungen zugeordnet sind. Aufgrund der Entkopplung der Container verfügt jeder Container über eine eigene Sitzungsverwaltung.
Wenn, wie in dem in Bild 22 dargestellten Fall, beide Container zur Realisierung einer J2EE Anwendung eingesetzt werden, dann sind die Daten, die der Sitzung eines Benutzers zuzurech-

nen sind, üblicherweise über beide Container verteilt. In Bild 22 wurde dies dadurch veranschaulicht, dass jedem Benutzer in jedem der beiden Container jeweils ein Akteur sowie ein Speicher für die im betreffenden Container untergebrachten Sitzungsdaten zugeordnet wurde. Außerdem liegt in diesem Fall eine ganz bestimmte Aufgabenverteilung zwischen den auf Basis dieser Container realisierten „user agents“ vor. Die eigentliche Datenverarbeitung im Sinne der „Business-Logik“ erfolgt in diesem Fall durch den „EJB Side User Agent“, während der „Web Side User Agent“ als Bindeglied zwischen Browser und „EJB Side User Agent“ dient. Dabei beschränkt sich die Aufgabe des „Web Side User Agents“ im Wesentlichen darauf, den „EJB Side User Agent“ von der Aufgabe zu befreien, die an der Benutzeroberfläche darzustellende Information in entsprechende HTML Seiten zu überführen.
Der zentrale Datenbestand, der mit Hilfe eines solchen Systems verwaltet wird, ist typischerweise in einer relationalen Datenbank abgelegt. Nun liegt der Codierung von Daten in relationalen Datenbanken eine Denkweise zugrunde, die sich von derjenigen, die objektorientierten Programmiersprachen zugrunde liegt, wesentlich unterscheidet. Dies wird in der Literatur auch gelegentlich als „impedance mismatch“ bezeichnet. Daher hat es sich bei der Formulierung von objektorientierten Programmen, welche der Verarbeitung von Daten dienen, die in relationalen Datenbanken abgelegt sind, in der Vergangenheit vielfach als zweckmäßig erwiesen, den Code zur Verarbeitung der Daten und den Code zum Zugriff auf die relationale Datenbank zu trennen. Diese Trennung beruht auf der Idee, die in der relationalen Datenbank abgelegten Daten für die Dauer der Verarbeitung in eine entsprechende Objektstruktur zu überführen, so dass die Verarbeitungsaufgabe als Bearbeitung dieser Objektstruktur formuliert werden kann. Der EJB Container unterstützt die Anwendung dieses Konzepts durch Bereitstellung eines entsprechenden Frameworks. Diejenigen Objekte, die dazu benutzt werden, Datenbankinhalte für die Dauer der Verarbeitung in Form einer Objektstruktur zugänglich zu machen, werden in der J2EE Spezifikation als „Entity Beans“ bezeichnet. Bei der Darstellung in Bild 22 wurde von der Nutzung dieses Frameworks ausgegangen. Der mit „Entity Beans“ bezeichnete Speicher enthält die Objektstruktur, die der Repräsentation von Daten aus der relationalen Datenbank dient. Um mit nebenläufigen Zugriffen auf die mittels „Entity Bean“-Objekten repräsentierten Datenbankinhalte sinnvoll umgehen zu können, unterstützt der Persistency Manager ein Transaktionskonzept, welches mit dem in heutigen Datenbanksystemen vorzufindenden vergleichbar ist. Der „Persistency Manager“ ist für den Abgleich der Datenbankinhalte mit den Objektattributen der „Entity Beans“ sowie für die Erzeugung und Zerstörung von „Entity Bean“ Objekten zuständig. Während die „EJB Side User Agents“ nur indirekt, über die vom „Persistency Manager“ verwalteten „Entity Beans“, auf Datenbankinhalte zugreifen, hat der „Persistency Manager“ direkten Zugang zum Datenbanksystem.

Im weiteren Verlauf dieses Abschnitts werden die anhand von Bild 22 eingeführten Begriffe aus der J2EE Welt weiter konkretisiert. Die Funktionsweise der beiden Container wird erst in den Abschnitten 3.3.1.2 und 3.3.1.3 näher vorgestellt. Dabei dient Abschnitt 3.3.1.2 der Vorstellung des „Web Containers“ und Abschnitt 3.3.1.3 der Vorstellung des „EJB Containers“. Um die in diesen beiden Abschnitten enthaltenen Aussagen besser veranschaulichen zu können, wird in dem nun folgenden Abschnitt 3.3.1.1 zunächst ein einfaches Beispiel einer J2EE Anwendung eingeführt, welches den in Bild 22 dargestellten typischen Aufbau aufweist. Diese Beispielanwendung ist Teil eines umfassenderen Beispielsystems, welches in der Literatur unter dem Namen „Pet Store“[20] bekannt ist und von der Firma Sun gewissermaßen als Vorbild für die Nutzung der J2EE Technologie entwickelt wurde.

20. Weitere Informationen hierzu sind auf folgender Web-Seite zu finden: „http://java.sun.com/blueprints/enterprise/“.

3.3.1.1 J2EE Beispielsystem „Pet Store“

Das „Pet Store“ System ist ein Informationssystem zur Unterstützung der Geschäftsprozesse einer Tierhandlung. Das Geschäftsmodell dieser Tierhandlung ist sehr einfach: Die Tierhandlung betreibt einen über das Internet erreichbaren „Online Shop“ und verfügt ansonsten über keinerlei Verkaufsräume. Die Kunden können mit Hilfe eines Web-Browsers auf den „Online Shop“ zugreifen, um sich das Sortiment der Tierhandlung anzusehen und Bestellungen an die Tierhandlung zu übermitteln. Die Tierhandlung züchtet selbst keine Tiere und betreibt auch keine eigene Vorratshaltung. Stattdessen kauft die Tierhandlung die entsprechenden Tiere erst beim Eintreffen einer Bestellung bei ihren Lieferanten ein, welche die entsprechenden Tiere im Namen der Tierhandlung direkt an die Kunden liefern.

Derjenige Systemteil des „Pet Store“ Systems, welcher den in Bild 22 dargestellten Grundaufbau hat, ist der „Online Shop“. Bild 23 zeigt die Einstiegsseite des „Online Shops“. Sie gliedert

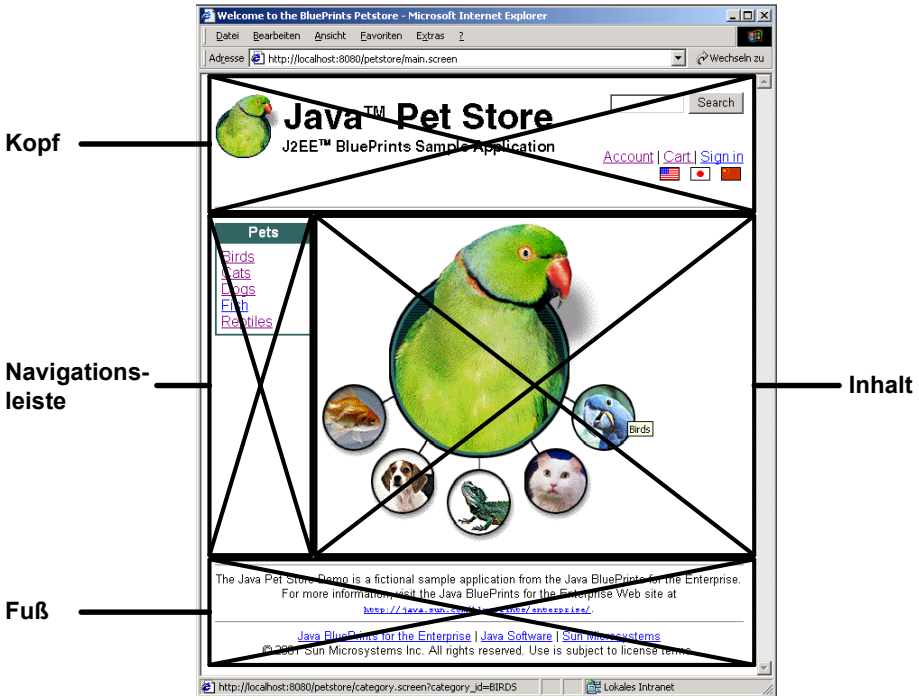


Bild 23: Web-Oberfläche der J2EE Beispielanwendung „Pet-Store“

sich, ebenso wie alle übrigen Seiten, die zur Realisierung der Benutzeroberfläche des „Online Shops“ dienen, in vier Bereiche[21]: Einen für alle Seiten identischen Kopf- und Fuß-Bereich, eine Navigationsleiste sowie einen mit „Inhalt“ bezeichneten Bereich. Die Navigationsleiste dient zur Navigation im Angebotskatalog der Tierhandlung. Bei der Navigation durch die im Katalog enthaltenen Angebote wird der mit „Inhalt“ bezeichnete Bereich zur Darstellung entsprechender Tabellen oder zur detaillierten Darstellung einzelner Angebote verwendet. Wenn ein Kunde an einem bestimmten Angebot interessiert ist, kann er den betreffenden Artikel in den ihm zur Verfügung gestellten „virtuellen Einkaufswagen“ hineinlegen. Der Kunde kann sich jederzeit den Inhalt seines virtuellen Einkaufswagens ansehen, darin enthaltene Artikel herausnehmen oder weitere im Katalog verzeichnete Artikel hinzufügen. Nachdem der Kunde seinen Einkaufswagen beladen hat, kann er die darin enthaltenen Artikel bestellen. Bevor die

21. Genau genommen gliedert sich die Benutzeroberfäche des „Online Shop“ sogar in sechs Bereiche, von denen zwei jedoch nur unter bestimmten Umständen sichtbar sind.

Übermittlung einer solchen Bestellung tatsächlich erfolgen kann, muss der Kunde sich entweder als Bestandskunde ausweisen oder er muss zunächst als Neukunde registriert werden. Bei der Registrierung als Neukunde wird er zur Eingabe seiner Personalien, der Lieferanschrift und seiner Kreditkartendaten aufgefordert. Die Darstellung derjenigen Teile der Bedienoberfläche, die zur Bearbeitung der kundenspezifischen Daten und des Einkaufwageninhalts dienen, erfolgt ebenfalls in dem mit „Inhalt“ bezeichneten Seitenbereich.

In Bild 22 wurde von dem häufig anzutreffenden Fall ausgegangen, dass die mit Hilfe eines J2EE basierten Systems verwalteten Daten in einer relationalen Datenbank abgelegt sind. Dies trifft auch auf das vorliegende „Pet Store“ Beispielsystem zu. Die „Pet Store“ Datenbank enthält folgende für den „Online Shop“ relevante Daten: Den Angebotskatalog der Tierhandlung inklusive Detailinformationen zu den einzelnen Tieren, die Personalien, Anschriften und Systemzugangsdaten der registrierten Kunden sowie deren Kreditkartendaten. Darüber hinaus nutzt das „Pet Store“ System diese Datenbank auch zur Ablage aller übrigen Unternehmensdaten wie zum Beispiel: eingegangene Bestellungen von Kunden, an Kunden gestellte Rechnungen, an Lieferanten herausgegebene Bestellungen, von Lieferanten vorliegende Rechnungen, Daten bezüglich der Kreditwürdigkeit der Kunden.

An der Bearbeitung der im Verlauf einer Benutzersitzung an den J2EE Server gestellten Requests sind im Fall des „Online Shops“ stets beide Container beteiligt. Dabei sind die Aufgaben zwischen dem „Web Side User Agent“ und dem „EJB Side User Agent“ so aufgeteilt, dass die eigentliche Verarbeitung von Daten im Sinne der „Business-Logik“ durch den „EJB Side User Agent“ erfolgt, während der „Web Side User Agent“ primär dazu dient, den „EJB Side User Agent“ von der oft aufwendigen Umwandlung der an der Benutzeroberfläche darzustellenden Information in entsprechende HTML Seiten zu befreien. Im vorliegenden Fall erstellt der „Web Side User Agent“, dem Design der Benutzeroberfläche folgend, bei jedem Request eine HTML Seite, welche sich in die angesprochenen vier Bereiche gliedert und in der vom Benutzer gewählten Sprache formuliert ist. Der typische Ablauf zur Bearbeitung einer HTTP-Anfrage durch den „Web Side User Agent“ besteht darin, dass er zunächst die Anfrage analysiert und die darin enthaltenen Parameter extrahiert. Wenn es sich um eine syntaktisch korrekte Anfrage handelt, erteilt er als nächstes einen Auftrag an den „EJB Side User Agent“. Als Ergebnis der Bearbeitung dieses Auftrags bekommt er vom „EJB Side User Agent“ ein Datenpaket, welches die Grundinformationen enthält, die er zur Berechnung der an den Browser zurückzusendenden HTML Seite benötigt.

Wenn ein Kunde der Tierhandlung beispielsweise gerade die Einstiegsseite des „Online Shops“ vor Augen hat und in der Navigationsleiste die Kategorie „Vögel“ auswählt, dann stellt der Browser einen HTTP Request an den J2EE Server. Wenn dieser Request beim J2EE Server eintrifft, wird er zunächst vom „Web Side User Agent“ geparst. Anschließend verpakt der „Web Side User Agent“ die in diesen HTTP Request enthaltenen Angaben praktisch unverändert in einen Request an den „EJB Side User Agent“. Das als Anwort auf diesem Request vom „EJB Side User Agent“ gelieferte Datenpaket enthält einerseits die Angabe, dass in dem mit „Inhalt“ bezeichneten Bereich der Bedienoberfläche als nächstes eine Liste von Angeboten darzustellen ist, und andererseits auch die für die Darstellung notwendigen Details zu den in dieser Liste darzustellenden Angeboten.

Die Liste der im virtuellen Einkaufswagen enthaltenen Artikel wird im vorliegenden Beispiel vom „EJB Side User Agent“ verwaltet. Da diese Daten nur für die Dauer der entsprechenden Benutzersitzung relevant sind, werden sie nicht in der Datenbank, sondern im „EJB Session Data“ Speicher abgelegt. Wenn ein Kunde den im „Kopf“-Teil enthaltenen Link zur Anzeige des aktuellen Einkaufwageninhalts anwählt, wird der entsprechende HTTP Request ganz ähnlich

wie im gerade beschriebenen Fall der Navigation im Angebotskatalog vom „Web Side User Agent“ nahezu unverändert an den „EJB Side User Agent“ weitergeleitet. Das daraufhin vom „EJB Side User Agent“ erstellte Antwortdatenpaket enthält die Angabe, dass in dem mit „Inhalt“ bezeichneten Bereich als nächstes die Seite zur Darstellung des Einkaufwageninhalts erscheinen soll, oder falls der Einkaufwagen leer ist, eine entsprechende Hinweisseite angezeigt werden soll. Falls der Einkaufwagen nicht leer ist, enthält das Antwortdatenpaket auch die Stückliste der im Einaufwagen enthaltenen Artikel und weitere Angaben, die sich auf das Angebot als Ganzes beziehen, beispielsweise: Gesamtbestellwert, Summe der Einzelpreise, gewährte Rabatte.

Wie bereits angesprochen, enthält die Datenbank unter anderem das Kundenverzeichnis der Tierhandlung. Zur Realisierung dieses Kundenverzeichnisses werden vier verschiedene Tabellen in der relationalen Datenbank verwendet: Customer, User, Address, Creditcard. Der „Online Shop“ erlaubt es, dass ein Kunde der Tierhandlung einen Großteil der für ihn spezifischen Daten direkt ändern kann. Bei der Bearbeitung der entsprechenden Requests greift der „EJB Side User Agent“ nicht direkt auf das relationale Datenbanksystem zu. Stattdessen nutzt er das vom „EJB Container“ bereitgestellte Framework und greift indirekt über entsprechende „Entity Bean“-Objekte auf die in diesen Tabellen enthaltenen Daten zu.
Im vorliegenden Fall ist die Abbildung zwischen den zum Zugriff auf die Datenbank verwendeten „Entity Bean“-Objekten und den in den Datenbanktabellen vorzufindenden Einträgen sehr einfach: Jedes „Entity Bean“-Objekt repräsentiert jeweils genau eine Zeile einer der vier genannten Tabellen. Für jede der vier Tabellen definiert die Implementierung des „Pet Store“ Systems jeweils eine eigene „Entity Bean“ Klasse. Die Klassendefinitionen folgen dabei dem Schema, dass sie für jede Spalte der korrespondierenden Tabelle ein Objektattribut definieren, welches zur Aufnahme entsprechender Spaltenwerte benutzt wird.
Wenn ein „EJB Side User Agent“ im Verlauf der Bearbeitung eines Requests Änderungen an Kundendaten vornehmen muss, wird er folgende Schritte durchführen: Zunächst meldet er beim Persistency Manager den Start einer Transaktion an. Danach bittet er den Persistency Manager um die Bereitstellung der benötigten „Entity Bean“-Objekte und modifiziert die Attribute dieser „Entity Bean“-Objekte so, wie es zur Bearbeitung des Requests erforderlich ist. Nach Durchführung aller zur Bearbeitung des Requests notwendigen Änderungen von Datenbankinhalten beendet er die zuvor begonnene Transaktion.

Im gerade beschriebenen Fall der Bearbeitung eines Requests zur Änderung der Kundendaten ist die Dauer einer Transaktion auf die Dauer der Bearbeitung eines Requests durch den „EJB-Container“ beschränkt. Eine derartige Beschränkung der Transaktionsdauer ist keineswegs durch die J2EE Spezifikation gefordert, und es ist durchaus zulässig, dass sich eine Transaktion über mehrere Requests erstreckt. Dennoch ist es für solche auf Basis der J2EE Technologie entwickelte Anwendungen typisch, dass sich die Dauer einer Transaktion nicht über mehrere Interaktionsschritte erstreckt. Häufig liegt sogar der geschilderte Fall vor, dass die Dauer einer solchen Transaktion auf die Dauer der Bearbeitung eines einzelnen an den „EJB Container“ gestellten Requests beschränkt ist.

3.3.1.2 Web Container

Bei dem im vorangegangenen Abschnitt beschriebenen „Online Shop“ Beispiel diente der „Web Container“ zur Realisierung der „Web Side User Agents“. Sämtliche Funktionalität, die diesen „Web Side User Agents“ zugeschrieben wurde, wurde bei der Implementierung des „Online Shop“ Beispiels in einer einzigen so genannten „Web Application“ definiert.
Der „Web Container“ ist ein Trägersystem für solche „Web Applications“. Dabei kann der glei-

che „Web Container“ für den Betrieb mehrerer „Web Applications“ benutzt werden. Die von einem „Web Container“ bereitgestellte Funktionalität weist gewisse Parallelen zu der eines Betriebssystems auf. Einerseits realisiert er Funktionen, die der Betriebsmittelverwaltung dienen, und andererseits bietet er Dienste an, die die Entwicklung von „Web Applications“ vereinfachen.
Der vom „Web Container“ bereitgestellte Funktionsumfang wird nun anhand von Bild 24 ge-

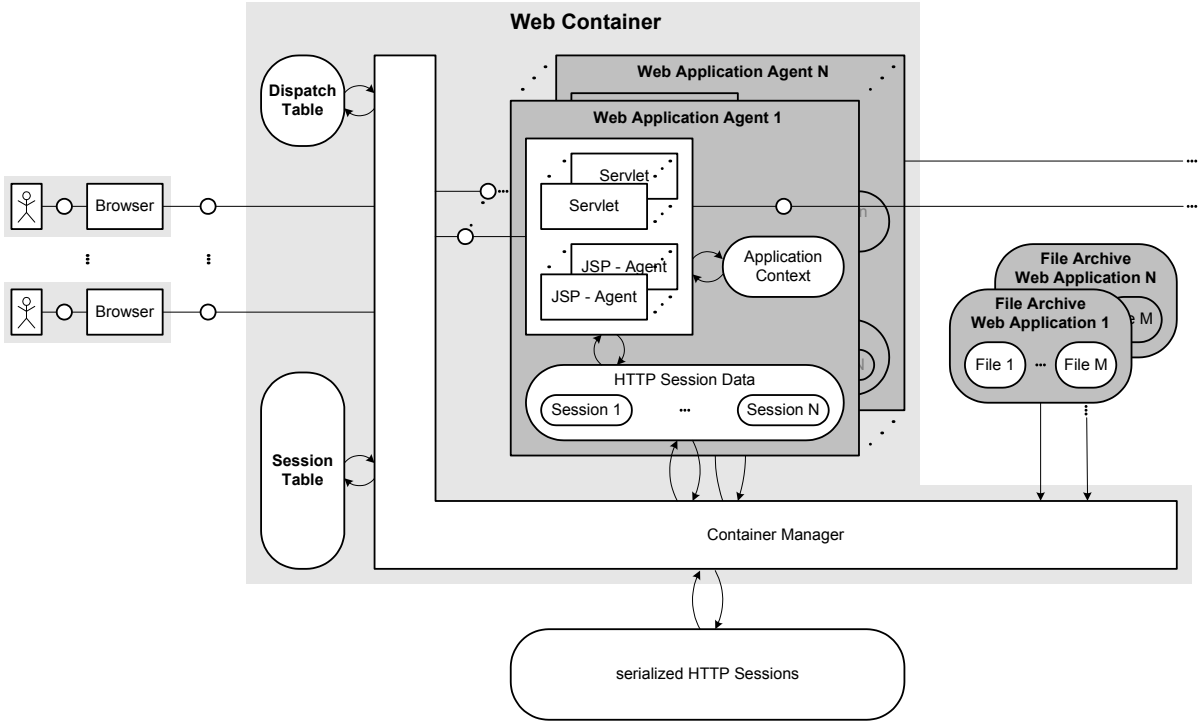


Bild 24: Aufbau des „Web Containers“

nauer vorgestellt. Links im Bild sind die Nutzer zu sehen, die mit Hilfe von Browsern auf den „Web Container“ zugreifen, um die dort installierten „Web Applications“ zu nutzen. Die vom „Web Container“ bereitgestellte Basisfunktionalität ist im Modell durch den als „Container Manager“ bezeichneten Akteur sowie durch die drei Speicher vertreten, die mit „Dispatch Table“, „Session Table“ und „serialized HTTP Sessions“ bezeichnet sind. Die „im“ Container installierten „Web Applications“ sind im Bild in zweifacher Weise vertreten. Zu jeder „Web Application“ gehört ein „Web Application Agent“ sowie ein „File Archive“. Die zu den „Web Applications“ gehörenden „File Archive“ Speicher sowie der mit „serialized HTTP Sessions“ beschriftete Speicher wurden explizit außerhalb des hellgrau hinterlegten „Web Container“ Bereichs dargestellt, da die in diesen Speichern enthaltenen Daten im Dateisystem abgelegt sind. Im Unterschied hierzu befinden sich die Komponenten, die innerhalb des hellgrau hinterlegten Bereichs dargestellt sind, „innerhalb“ der zum Betrieb des „Web Containers“ benutzten Java VM. Dementsprechend befinden sich die Inhalte der Speicher „Dispatch Table“, „Session Table“, „Application Context“ und „HTTP Session Data“ letztlich im Hauptspeicher.

Dispatch von Anfragen

Als Antwort auf die im Rahmen der Nutzung einer Anwendung an den „Web Container“ gestellten Anfragen werden typischerweise Daten zurückgeliefert, welche der Beschreibung von HTML Seiten dienen. Zunächst einmal gibt es dabei den für die Realisierung interaktiver Anwendungen typischen Fall, dass die infolge eines Requests zurückgelieferten Antwortdaten erst im Verlauf der Bearbeitung des Requests berechnet werden. Im einfachsten Fall besteht die Be-

arbeitung eines Requests jedoch nur darin, dass der Inhalt einer bestimmten Datei zurückgeliefert wird. Diese beiden unterschiedlichen Arten der Bearbeitung von Requests spiegeln sich auch in Bild 24 wider. Im Innern der „Web Application Agents“ sind einerseits „Servlets“ und „JSP Agents“ dargestellt, welche vom „Container Manager“ mit der Bearbeitung von eintreffenden Requests beauftragt werden können. Andererseits gehört zu jeder „Web Application“ ein „File Archive“, dessen Inhalt im direkten Zugriff des „Container Managers“ liegt. Den Inhalt dieser Dateien sendet der „Container Manager“ beim Eintreffen entsprechender Requests als Antwort an den jeweiligen Browser zurück.
Eine wichtige Aufgabe des „Container Managers“ besteht darin, die eintreffenden Requests zu analysieren und sie an die entsprechenden „Servlets“ und „JSP Agents“ der verschiedenen „Web Applications“ weiterzuleiten. Die Entscheidung darüber, wer für die Bearbeitung eines Requests zuständig ist, trifft der „Container Manager“ auf Basis der im Request enthaltenen URL sowie der Information, die in dem als „Dispatch Table“ bezeichneten Speicher abgelegt ist. Die URLs sämtlicher Requests, die an eine bestimmte „Web Application“ adressiert sind, weisen ein für die betreffende „Web Application“ spezifisches Präfix auf, welches bei der Installation der „Web Application“ im „Web Container“ festgelegt wird. Hier zwei Beispiele für solche URLs:

(1) „http://localhost:8080/petstore/main.screen“
(2) „http://localhost:8080/petstore/images/banner_logo.gif“

Die erste der beiden URLs ist diejenige URL, die zum Aufruf der in Bild 23 dargestellten „Online Shop“ Einstiegsseite verwendet wurde. Als Antwort auf diese Anfrage bekommt der Browser eine von einem „Servlet“ berechnete HTML-Seite. Diese vom „Servlet“ berechnete HTML-Seite enthält eine Vielzahl von Verweisen auf Bilder, jedoch nicht die zur Darstellung notwendigen Bilddaten. Daher muss der Browser zur Darstellung der Seite weitere Requests an den „Web Container“ richten, um an die entsprechenden Bilddaten zu gelangen. Bei der zweiten URL handelt es sich um eine URL, die zur Anforderung der Bilddaten des im Kopf der „Online Shop“-Startseite angezeigten „Java™ Pet Store“-Banners dient. Solche Requests, die der Anforderung von Bilddaten dienen, werden im Fall des „Online Shops“ grundsätzlich direkt durch den „Container Manager“ beantwortet, indem er den Inhalt einer entsprechenden Datei als Ergebnis an den Browser zurücksendet.
Die beiden aufgelisteten URLs gehören zu Requests, die an die gleiche „Web Application“ adressiert waren und haben daher das gemeinsame Präfix „http://localhost:8080/petstore/“. Die den verschiedenen „Web Applications“ zugeordneten Präfixe bilden nur einen Teil der Information, die vom „Container Manager“ benötigt wird, um über die weitere Verfahrensweise zur Bearbeitung von Requests zu entscheiden. Dementsprechend wird im Rahmen der Installation einer „Web Application“ nicht nur das ihr zugeordnete Präfix in der „Dispatch Table“ eingetragen. Vielmehr wird dort detailliert verzeichnet, welche „Servlets“ und „JSP Agents“ mit der Beantwortung von Anfragen bezüglich bestimmter URLs beauftragt werden sollen. Außerdem wird in der „Dispatch Table“ auch der Name des Wurzelverzeichnisses eingetragen, welches das der „Web Application“ zugeordnete „File Archive“ beinhaltet. Der Algorithmus, den der „Container Manager“ auf eine URL anwendet, um zu entscheiden, wer für die Bearbeitung des Requests zuständig ist, ist Folgender: Zunächst überprüft der „Container Manager“, ob eine „Web Application“ installiert ist, der ein Präfix zugeordnet wurde, welches mit dem der gegebenen URL übereinstimmt. Wenn er eine solche „Web Application“ findet, überprüft er, ob ein bestimmter „Servlet“-Akteur oder ein „JSP Agent“ für diese URL registriert ist und leitet den betreffenden Request gegebenenfalls an den dafür registrierten Akteur weiter. Wenn das Präfix der URL zu einer installierten „Web Application“ passt und kein zuständiger „Servlet“-Akteur oder „JSP Agent“ gefunden wird, interpretiert der „Container Manager“ die Zeichenfolge, die

nach dem Präfix folgt, als einen zum Wurzelverzeichnis des „File Archive“ der betreffenden „Web Application“ relativen Dateipfad und liefert den Inhalt der so identifizierten Datei als Antwort an den Browser zurück.

Sitzungsverwaltung

Ein „Web Application Agent“ besteht aus Sicht des Containers aus einer Menge von „Servlets“ und „JSP Agents“, welche Zugriff auf die beiden als „Application Context“ und „HTTP Session Data“ bezeichneten Speicher haben. Der „Application Context“ liegt dabei ausschließlich im Zugriff der „Web Application“-Akteure, und seine innere Struktur ist dem „Container Manager“ nicht bekannt. Auf den Inhalt des „HTTP Session Data“ Speichers hat hingegen auch der „Container Manager“ Zugriff. Dieser Speicher ist speziell für die Ablage von Sitzungsdaten vorgesehen. Dass auch der „Container Manager“ Zugriff auf diesen Speicher hat, liegt daran, dass er einen Teil der mit der Verwaltung von Benutzersitzungen in Zusammenhang stehenden Aufgaben übernimmt. Insbesondere bietet er einen Mechanismus zur Auslagerung von Sitzungsdaten an. Dieser „Auslagerungsmechanismus“ dient der Speichervirtualisierung und ermöglicht es, dass mehr Sitzungsdaten verwaltet werden können als in den Heap der zum Betrieb des „Web Containers“ eingesetzten Java VM „hineinpassen“ würden. Als Ablageort für diejenigen Sitzungsdaten, die aktuell nicht durch entsprechende Objekte im Heap der Java VM repräsentiert sind, dient der mit „serialized HTTP Sessions“ bezeichnete Speicher. Die Auslagerung von Sitzungsdaten erfolgt so, dass sämtliche einer Sitzung zugeordneten Daten als Ganzes ausgelagert und später wieder als Ganzes eingelesen werden. Der mit „Session Table“ bezeichnete Speicher enthält ein Verzeichnis aller existierenden Sitzungen. In ihm ist auch vermerkt, welche Sitzungsdaten sich aktuell im Hauptspeicher befinden und welche ausgelagert sind.
An dieser Stelle muss auf den Umstand hingewiesen werden, dass die Nutzung des HTTP Protokolls als Basis für die Kommunikation zwischen Browsern und den auf Basis des „Web Containers“ realisierten „Web Side User Agents“ mit verschiedenen Problemen verbunden ist. Die Ursache für diese Probleme liegt darin, dass das HTTP Protokoll als so genanntes „zustandsloses“ Protokoll konzipiert wurde. Dementsprechend ist ein HTTP Server auf Basis der in der Protokollspezifikation getroffenen Festlegungen nicht in der Lage, Zusammenhänge zwischen verschiedenen an ihn gerichteten Anfragen zu erkennen. Für die Entwicklung von interaktiven Anwendungen ist es jedoch notwendig, die Zugehörigkeit zwischen den von den Browsern kommenden Requests und den vorhandenen Benutzersitzungen feststellen zu können. Da diese Funktionalität von grundlegender Bedeutung für die Entwicklung von interaktiven Anwendungsprogrammen ist, ist sie ein fester Bestandteil des vom „Web Container“ bereitgestellten Funktionsumfangs. Zur Realisierung dieser Funktionalität können zwei unterschiedliche Techniken eingesetzt werden. Eine Möglichkeit zur Realisierung dieser Funktionalität ist unter der Bezeichnung „URL-Rewriting“ bekannt, die zweite Möglichkeit basiert auf der Nutzung so genannter „Cookies“.

- URL-Rewriting
  Das dem URL-Rewriting zugrunde liegende Prinzip geht von der Vorstellung aus, dass jede Benutzersitzung mit einem ersten Request beginnt und dass in sämtlichen nachfolgenden Requests, die dieser Benutzersitzung zuzurechnen sind, nur solche URLs verwendet werden, die in den Antwortdaten vorangegangener Requests enthalten waren. Das Konzept des URL-Rewriting besteht nun darin, sämtliche URLs, die an den HTTP-Client eines bestimmten Benutzers übermittelt werden, um einen Identifikator für die zugehörige Benutzersitzung zu erweitern. Auf diese Weise kann der Web-Container die für die Bearbeitung eines Requests relevante Benutzersitzung direkt an der URL „ablesen“.

- Cookies
  Der Begriff Cookies steht mit der in [RFC 2109-Cookies] beschriebenen Erweiterung des HTTP-Protokolls in Zusammenhang. Cookies können von einem HTTP-Server erzeugt und als Bestandteil der Antwort auf einen HTTP-Request an HTTP-Clients übertragen werden. HTTP-Clients, welche Cookies unterstützen, sammeln diese Cookies, um sie den Requestdaten nachfolgender HTTP-Requests hinzuzufügen. Dabei fügen sie einem HTTP-Request stets genau diejenigen Cookies hinzu, die sie von dem HTTP-Server bekommen haben, an den der Request gerichtet ist. Web Container nutzen Cookies, um einen Identifikator für die zugehörige Benutzersitzung darin zu speichern. Der Cookie mit dem Identifikator für die Benutzersitzung wird dabei üblicherweise beim ersten HTTP-Request eines neuen Benutzers erzeugt. Da der Cookie bei den nachfolgenden Requests vom HTTP-Client immer wieder mitgeliefert wird, kann der Container Manager die nachfolgenden Requests der betreffenden Benutzersitzung zuordnen.

Ein weiteres Problem, das mit der Nutzung des HTTP-Protokolls in Zusammenhang steht, besteht darin, dass der Container Manager nicht ohne weiteres feststellen kann, wann eine Benutzersitzung zu Ende ist. Um zu verhindern, dass sich im Web-Container Daten ansammeln, die zu Benutzersitzungen gehören, deren Ende nicht sicher festgestellt werden konnte, verwirft der Container Manager alte Sitzungsdaten automatisch. Dabei werden die Daten einer Sitzung genau dann verworfen, wenn nicht innerhalb eines gewissen Zeitraums weitere Anfragen eintreffen, die der betreffenden Sitzung zuzurechnen sind.

Servlets und JSP-Agents

Bisher wurde noch nicht auf den Unterschied zwischen Servlets und JSP-Agents eingegangen. Der wesentliche Unterschied zwischen diesen beiden Arten von Requestbeantwortungsakteuren besteht darin, wie die durch sie realisierte Funktionalität beschrieben wird.

Die Funktionalität eines Servlet-Akteurs wird in Form einer Java Klasse beschrieben. Der Ablauf zur Bearbeitung von Requests wird dabei vom Programmierer der Servlet-Klasse in Form einer Java-Methode definiert, welche vom Container für jeden Request, der durch dieses Servlet bearbeitet werden soll, einmal aufgerufen wird. Beim Aufruf der Requestbearbeitungsmethode übergibt der Container verschiedene Parameter an die Methode. Mit Hilfe der als Parameter übergebenen Java-Objekte kann im Verlauf der Methodenausführung sowohl auf die Requestdaten als auch auf die Daten der Benutzersitzung zugegriffen werden, zu der der betreffende Request gehört. Ferner wird auch ein Parameterobjekt übergeben, welches zur Übermittlung der Antwortdaten an den HTTP-Client dient. Neben der Methode, in der der Ablauf zur Bearbeitung eines Requests beschrieben ist, kann in der Servlet-Klasse noch eine „init()“- und eine „destroy()“-Methode definiert werden. Die „init()“-Methode wird vom Container Manager aufgerufen, bevor er das Servlet mit der Bearbeitung des ersten Requests beauftragt. Analog wird die „destroy()“-Methode eines Servlets aufgerufen, wenn die Web-Application außer Betrieb genommen wird. Ein typisches Beispiel für solche Funktionalität, die in der „init()“-Methode einer Servlet-Klasse codiert ist, ist diejenige, die der Initialisierung des „Application Context“ Speichers dient. Im Fall der Pet-Store „Online Shop“ Anwendung besteht die Initialisierung des „Application Contextes“ darin, verschiedene Parameterdateien einzulesen und deren Inhalt in Form entsprechender Java-Objekte im „Application Context“ Speicher abzulegen.

Es ist für Web-Applications typisch, dass die Antwort auf einen Request aus Daten besteht, die eine vom HTTP-Client (Browser) darzustellende HTML-Seite beschreiben. Da in der Requestbearbeitungsmethode einer Servlet-Klasse beliebige Abläufe formuliert werden können, hat der Anwendungsprogrammierer die Möglichkeit, beliebige HTML-Seiten zu berechnen und als

Antwort an den HTTP-Client zu senden. Die Nutzung von Servlets zur Berechnung von HTML-Seiten ist jedoch mit einem großen Nachteil verbunden, der in der Folge zur Entwicklung der „JSP“-Technologie geführt hat. Dieser Nachteil besteht darin, dass der in der Requestbearbeitungsmethode eines Servlets enthaltene Java-Code normalerweise nur schwer lesbar ist, weil er mit den Zeichenketten, aus denen die zu erzeugenden HTML-Seiten zusammengesetzt werden, durchsetzt ist. Das Ziel der Entwicklung der „JSP“-Technologie bestand daher darin, die Durchmischung von „HTML-Zeichenketten“ und Java-Programmcode zu vermeiden. Um dieses Ziel zu erreichen, verfolgt die „JSP“-Technologie den Ansatz, die zu erzeugenden HTML-Seiten deklarativ zu beschreiben. Eine solche deklarative Beschreibung wird als „**J**ava **S**erver **P**age (JSP)“ bezeichnet. Die einfachste Form einer „JSP“-Seite liegt vor, wenn in der betreffenden „JSP“-Seite gar keine „variablen Anteile“ definiert werden. In diesem Fall ist die Definition der „JSP“-Seite mit dem Inhalt der auf Basis dieser „JSP“-Seite erzeugten HTML-Seiten identisch. Im Allgemeinen enthalten „JSP“-Seiten neben den „HTML-Zeichenketten“, die unverändert in die zu erzeugenden HTML-Seiten übernommen werden, auch noch so genannte „Tags“. Durch Verwendung solcher Tags kann der Entwickler der „JSP“-Seite einerseits Platzhalter definieren, die im Verlauf der Requestbearbeitung substituiert werden. Andererseits kann mit Hilfe von „Schleifen“- und „Fallunterscheidungs“-Tags auch spezifiziert werden, dass bestimmte Teile der betreffenden JSP-Seite im Verlauf der Requestbearbeitung mehrfach oder nur unter bestimmten Bedingungen berücksichtigt werden.
Der mit der „JSP“-Technologie verfolgte Ansatz ist sehr nahe liegend, wenn man bedenkt, dass sich die Benutzeroberfläche eines Anwendungsprogramms typischerweise aus einem bestimmten Repertoire an „Bildschirmseiten“ mit vorab festgelegtem Aufbau zusammensetzt. Im Fall des Pet-Store „Online Shops“ ist die gesamte Benutzeroberfläche in Form von 25 „JSP“-Seiten beschrieben. Dabei ist es jedoch nicht so, dass jede „Bildschirmseite“ des „Online Shops“ durch genau eine „JSP“-Seite beschrieben ist. Stattdessen sind für die Beschreibung einer „Bildschirmseite“ im Allgemeinen mehrere „JSP“-Seiten relevant. Beispielsweise sind für Erzeugung der in Bild 23 dargestellten Einstiegsseite des „Online Shops“ insgesamt fünf „JSP“-Seiten relevant. Eine dieser „JSP“-Seiten dient dazu, die Aufteilung der Seite in vier Bereiche zu definieren. Die anderen vier „JSP“-Seiten dienen der Definition der in diesen vier Bereichen dargestellten Seiten(-teile). Während diejenige „JSP“-Seite, die zur Definition des mit Inhalt bezeichneten Bereichs der Einstiegsseite dient, ausschließlich für die Einstiegsseite des „Online Shops“ relevant ist, sind die anderen vier „JSP“-Seiten zugleich auch für die übrigen „Bildschirmseiten“ des „Online Shops“ relevant. In Bild 25 sind zwei weitere Beispielseiten abgebildet, die in dem mit Inhalt bezeichneten Bereich der Benutzeroberfläche des „Online Shops“ dargestellt werden können. Links im Bild ist eine detaillierte Darstellung eines Angebots aus

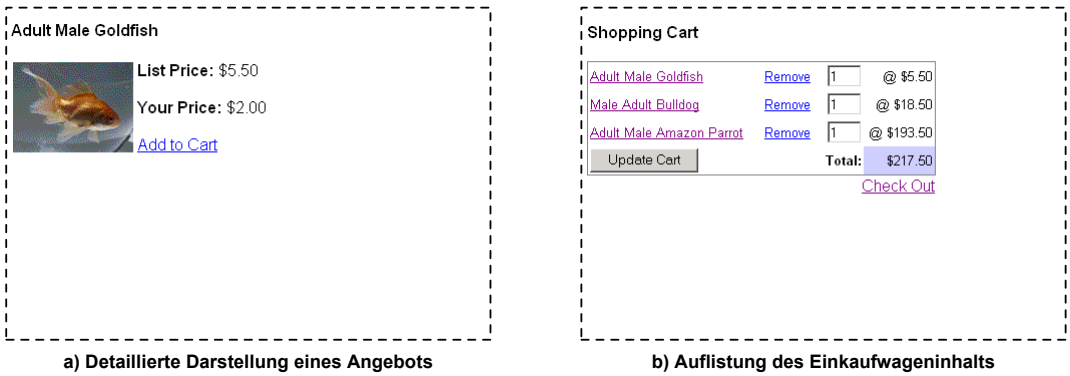


Bild 25: Beispiele für HTML-Seiten, die mittels Java Server Pages beschrieben wurden

dem Katalog der Tierhandlung zu sehen. In der rechten Bildhälfte ist eine Seite dargestellt, auf

der der Inhalt eines virtuellen Einkaufwagens aufgelistet ist. Auch diese beiden Seiten der Benutzeroberfläche sind durch je eine JSP-Seite beschrieben. Während die „JSP“-Seite, aus der die links dargestellte Seite erzeugt wurde, außer „HTML-Zeichenketten“ nur „Platzhalter“-Tags enthält, wurde bei der Definition der „JSP“-Seite, aus der die rechts dargestellte Seite erzeugt wurde, auch ein „Fallunterscheidungs“- und ein „Schleifen“-Tag genutzt. Das „Fallunterscheidungs“-Tag, wird benötigt um festzulegen, dass bei der Darstellung des Inhalts eines leeren Einkaufwagens keine tabellarische Auflistung erfolgt, sondern nur ein Hinweistext erscheinen soll. Die Spezifikation der tabellarischen Auflistung ist mit Hilfe eines „Schleifen“-Tags definiert, wobei der Abschnitt, der den Rumpf dieses „Schleifen“-Tags darstellt, die Spezifikation einer einzelnen Tabellenzeile enthält.
Das Repertoire an Tags, das bei der Definition von „JSP“-Seiten verwendet werden kann, ist nicht in der J2EE Spezifikation festgelegt, sondern wird letztlich vom Entwickler des jeweiligen Anwendungsprogramms definiert. Die Definition eines Tag-Typs geschieht dabei in Form einer Java Klasse. Da eine detaillierte Vorstellung der „JSP“-Technologie an dieser Stelle zu weit führen würde, soll hier nur noch kurz darauf hingewiesen werden, dass die „JSP“-Seiten bei der Installation der betreffenden Web-Application im Web-Container mit Hilfe eines „JSP-Compilers“ übersetzt werden. Dabei wird aus jeder „JSP“-Seite eine Servlet-Klasse generiert. Die Leistung des „JSP-Compilers“ besteht dabei im Wesentlichen darin, die in der betreffenden „JSP“-Seite enthaltenen Angaben in eine entsprechende Implementierung der erzeugten Requestbearbeitungsmethode zu überführen.

3.3.1.3 Enterprise Java Bean Container

Die im letzten Abschnitt vorgestellte „Web Container“ Technologie legte in gewisser Weise den Grundstein für die Entwicklung der gesamten J2EE Technologie und wurde bereits lange vor der Existenz des „EJB-Containers“ zur Realisierung unternehmensweiter Informationssysteme eingesetzt. Die steigende Komplexität der auf Basis des „Web Containers“ entwickelten Systeme war letztlich die Ursache für die Entwicklung der „EJB Container“ Technologie. Ähnlich wie der „Web Container“ stellt auch der „EJB-Container“ Trägersystemfunktionalität zur Verfügung. Die im „EJB-Container“ installierbaren Anwendungskomponenten werden dabei als „**E**nterprise **J**ava **B**eans (EJBs)“ bezeichnet. Insgesamt gibt es drei Arten solcher EJBs: „Session Beans“, „Entity Beans“ und „Message Driven Beans“.
Im Folgenden wird der Verwendungszweck sowie die vom „EJB-Container“ angebotene Unterstützung für die verschiedenen Arten von Beans genauer vorgestellt.

Session Beans

Obwohl die Bezeichnung „Session Bean“ bisher nicht verwendet wurde, finden sich in der Implementierung des Pet-Store Systems eine Reihe von Anwendungsbeispielen für Session Beans. So werden sie beispielsweise im Kontext des „Online Shops“ zur Realisierung der Funktionalität der „EJB Side User Agents“ eingesetzt. Jeder „EJB Side User Agent“ wird dabei durch mehrere „Session Beans“ unterschiedlichen Typs realisiert. Ein Beispiel hierfür ist der als „Shopping Cart EJB“ bezeichnete „Session Bean“ Typ. Ein „Session Bean“ von diesem Typ dient der Verwaltung des virtuellen Einkaufwagens eines „Online Shop“ Benutzers.
Das gerade genannte Anwendungsbeispiel weist ein Charakteristikum auf, das für den Einsatz von „Session Beans“ sehr typisch ist. Dieses Charakteristikum besteht darin, dass ein solches „Shopping Cart EJB“ einer Benutzersitzung zugeordnet ist. Wenn ein Benutzer mit der Nutzung der „Online Shop“ Anwendung beginnt, wird das ihm zugeordnete „Shopping Cart EJB“ erzeugt, und es wird wieder zerstört, wenn die Sitzung dieses Benutzers endet.

Bevor näher auf die Verwendung von „Session Beans“ im Pet-Store System eingegangen wird, soll das generelle Konzept anhand von Bild 26 eingeführt werden. „Session Beans“ sind spezielle vom „EJB Container“ verwaltete Objekte. Ihre Erzeugung muss explizit beim „Container Manager“ in Auftrag gegeben werden. Dabei wird die Erzeugung typischerweise durch den späteren Nutzer des „Session Beans“ in Auftrag gegeben. Als Ergebnis erhält der Nutzer einen Identifikator für das erzeugte „Session Bean“, welchen er in der Folge dazu benutzen kann, Aufträge an das „Session Bean“ zu erteilen. Ein solches „Session Bean“ verfügt über einen Zustandsspeicher, um Daten über die Dauer der Bearbeitung eines Auftrags hinaus aufzubewahren.

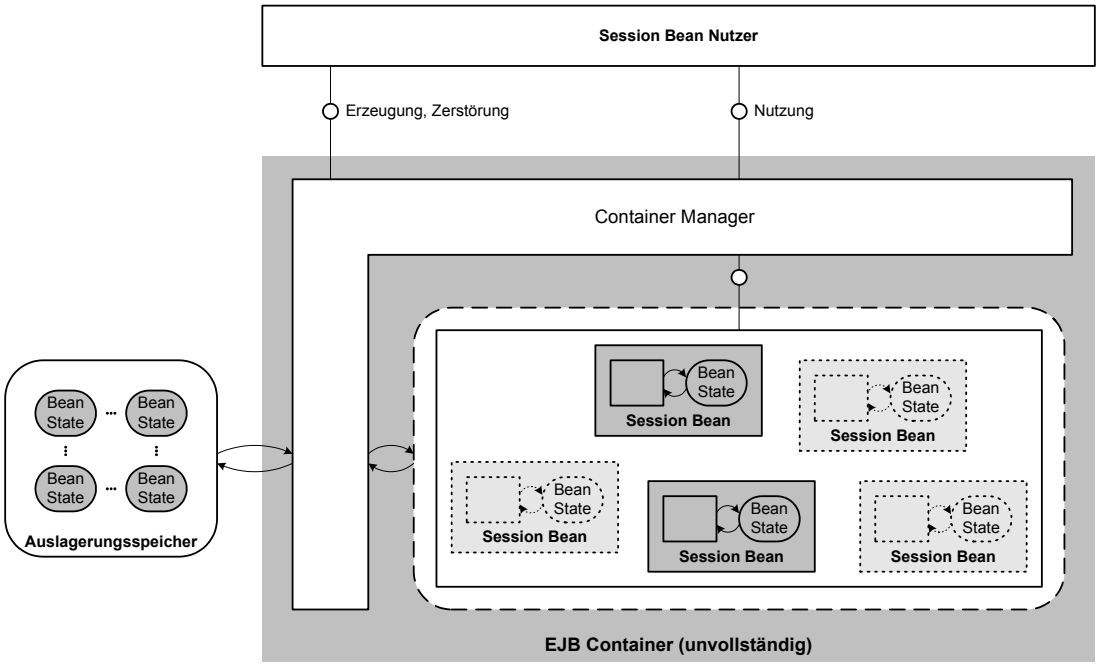


Bild 26: Session Beans - Grundkonzept

Wie bereits angesprochen, hat der EJB Container unter anderem die Aufgabe, für eine sinnvolle Nutzung der vorhandenen Betriebsmittel zu sorgen. Aus diesem Grund stellt er einen Auslagerungsmechanismus für „Session Beans“ bereit, der es ermöglicht, dass nur ein Teil der jeweils existierenden „Session Beans“ in Form von entsprechenden Java-Objekten im Hauptspeicher gehalten werden muss. Daher ist im Bild 26 ein Auslagerungsspeicher dargestellt, in dem die für die Rekonstruktion der ausgelagerten „Session Beans“ notwendigen Daten abgelegt werden. Typischerweise ist dieser Auslagerungsspeicher mit Hilfe von Dateien realisiert. Einige Applicationserver bieten darüber hinaus auch die Möglichkeit, hierfür ein Datenbanksystem einzusetzen. Die innerhalb des EJB Containers nur angedeuteten „Session Beans“ befinden sich im Auslagerungszustand.
Zu jedem „Session Bean“, welches aktuell nicht ausgelagert ist, gibt es ein entsprechendes Java Objekt innerhalb der zur Realisierung des EJB Containers dienenden Java-VM. Der Typ dieses Java-Objekts definiert die Eigenschaften des betreffenden Beans. Daher besteht die Entwicklung eines neuen Bean Typs im Wesentlichen in der Entwicklung einer entsprechenden Java Klasse. Die Erteilung eines Auftrags an ein „Session Bean“ führt letztlich dazu, dass eine Methode des zur Realisierung des betreffenden „Session Bean“ dienenden Java-Objektes aufgerufen wird. Die Aufgabe des Containers ist jedoch nicht auf die Durchführung dieses Methodenaufrufs beschränkt.
Ein Teil der Eigenschaften eines Session Beans werden vom Bean Entwickler nämlich nicht in die betreffende Java-Klasse „hineinprogrammiert“, sondern deklarativ festgelegt. Diese Dekla-

rationen werden in einer XML-Datei definiert, welche als „deployment descriptor“ bezeichnet wird. Im „deployment descriptor“ kann beispielsweise definiert werden, dass bestimmte Methoden eines Session Beans nur dann aufgerufen werden dürfen, wenn dieser Aufruf im Rahmen der Sitzung eines Benutzers durchgeführt wird, der einer bestimmten Benutzergruppe angehört. Weiterhin können über den „deployment descriptor“ auch Start- und Ende-zeitpunkte von (Datenbank-) Transaktionen deklarativ festgelegt werden. So kann man beispielsweise festlegen, dass vor der Ausführung einer bestimmten Methode eines Session Beans eine Transaktion begonnen werden muss, und dass diese Transaktion nach der Ausführung der Methode wieder zu beenden ist.
Um die im „deployment descriptor“ definierten Regeln umzusetzen, führt der Container sowohl vor als auch nach der Durchführung von Session Bean Methodenaufrufen entsprechende Verarbeitungsschritte durch.

Bereits zu Beginn dieses Abschnitts wurde darauf hingewiesen, dass ein auf Basis der J2EE Technologie hergestelltes System im Allgemeinen über mehrere Rechner/Java VM’s verteilt ist. Anhand von Bild 27 soll nun erläutert werden, welche Unterstützung der EJB Container für

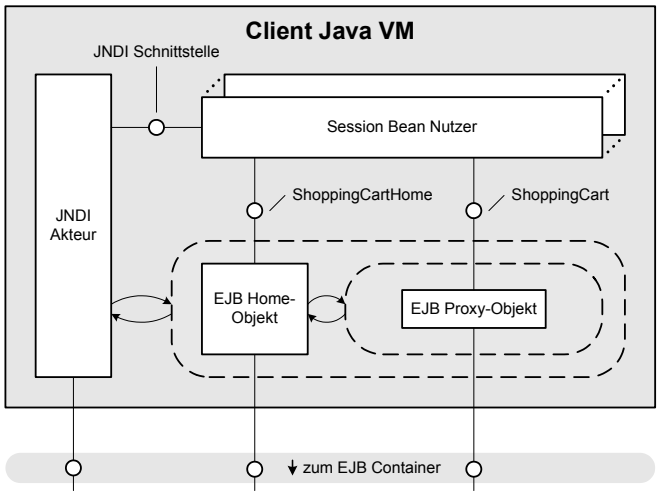


Bild 27: Nutzung von Session Beans aus Entwicklersicht

den „Fernzugriff“ auf die im EJB Container beherbergten „Session Beans“ bietet. Das Bild zeigt eine Java VM, in deren Innern verschiedene Akteure dargestellt sind. Diese Akteure nutzen Netzwerkverbindungen, um auf „Session Beans“ zuzugreifen. Die Netzwerkverbindungen sind in Form von Kanälen am unteren Bildrand dargestellt. Die genutzten „Session Beans“ sind im Bild nicht dargestellt. Stattdessen soll angenommen werden, dass der entsprechende EJB Container über die am unteren Bildrand dargestellten Netzwerkverbindungen erreichbar ist.
Für jedes „Session Bean“, welches von Programmkomponenten innerhalb der „Client Java VM“ genutzt wird, existiert in der „Client Java VM“ ein entsprechendes Stellvertreter-Objekt. Darüber hinaus existiert auch für jeden genutzten „Session Bean“-Typ jeweils ein Stellvertreter-Objekt in der „Client Java VM“. Um die Übersichtlichkeit zu wahren, wurde in Bild 27 nur ein Stellvertreterobjekt von jedem dieser beiden Typen dargestellt. Das „EJB Proxy-Objekt“ ist dabei ein Stellvertreter-Objekt für ein „Session Bean“, während das „EJB Home-Objekt“ einen „Session Bean“ Typ vertritt.
Die „EJB Proxy-Objekte“ bieten Schnittstellen an, welche von den „Session Bean Nutzern“ zur Übermittlung von Aufträgen an die korrespondierenden „Session Beans“ verwendet werden können. Dabei sind die Schnittstellen der „EJB Proxy-Objekte“ vom Typ des jeweils repräsentierten „Session Beans“ abhängig. Beispielsweise bieten Proxy-Objekte für „Session Beans“ vom Typ „Shopping Cart EJB“ eine „addItem()“-Methode an, um dem Inhalt des virtuellen Ein-

kaufwagens, der durch das korrespondierende „Shopping Cart EJB“ implementiert ist, einen Artikel hinzuzufügen.
Während die „EJB Proxy-Objekte“ zur Kommunikation mit bereits existierenden „Session Beans“ dienen, sind die „EJB Home-Objekte“ für die Erzeugung neuer „Session Beans“ und „EJB Proxy-Objekte“ von Bedeutung. In der Mehrzahl der Fälle wird die Erzeugung eines „Session Bean“ durch den späteren „Session Bean Nutzer“ veranlasst. Solche Erzeugungsaufträge richten die „Session Bean Nutzer“ an das für den gewünschten „Session Bean“-Typ zuständige „EJB Home-Objekt“. Das „EJB Home-Objekt“ leitet den Auftrag zur Erzeugung des neuen „Session Beans“ an den EJB Container weiter, welcher als Ergebnis einen Identifikator für das neu erzeugte „Session Bean“ an das „EJB Home-Objekt“ zurückliefert. Nach Erhalt dieses Identifikators, erzeugt das „EJB Home-Objekt“ ein neues „EJB Proxy-Objekt“ welches in der Folge als Repräsentant des neu erzeugten „Session Beans“ dient.
Neben dem Fall, dass ein „EJB Proxy-Objekt“ zusammen mit dem zugehörigen „Session Bean“ durch das zuständige „EJB Home-Objekt“ erzeugt wird, besteht auch die Möglichkeit, „EJB Proxy-Objekte“ für bereits existierende „Session Beans“ zu erzeugen. Diese Situation, dass ein „EJB Proxy-Objekt“ für bereits existierende „Session Beans“ erzeugt werden soll, tritt typischerweise im Zusammenhang mit der Nutzung des „Java Naming and Directory Service“ auf. Der „Java Naming and Directory Service“ ist ein Ablagesystem, welches zum Austausch von Konfigurationsinformation zwischen den verschiedenen Programmkomponenten eines J2EE basierten Systems dient. Auf die darin abgelegten Daten kann praktisch von allen Komponenten einer J2EE Anwendung zugegriffen werden. Dies gilt sowohl für Komponenten, die im Web- oder EJB-Container beherbergt sind, als auch für solche Komponenten, die durch „gewöhnliche“ Java Programme hergestellt sind. Im Innern der im Bild 27 dargestellten „Client Java VM“ ist ein „JNDI Akteur“ dargestellt. Dieser Akteur ist Bestandteil des „Java Naming and Directory Service“ und bietet eine „**J**ava **N**aming and **D**irectory **I**nterface (**JNDI**)“ Schnittstelle zum Zugriff auf die vom „Java Naming and Directory Service“ verwalteten Daten an. Die Identifikation der vom „Java Naming and Directory Service“ verwalteten Einträge erfolgt mittels Zeichenketten wie zum Beispiel "java:comp/env/ejb/ShoppingCartHome". Wenn einer der im Bild dargestellten „Session Bean Nutzer“ eine Anfrage an den „Java Naming and Directory Service“ stellt, bedeutet dies letztlich, dass eine Methode eines Java-Objektes aufgerufen wird, das der Implementierung des „JNDI Akteurs“ zuzurechnen ist. Das im Rahmen der Bearbeitung einer solchen Anfrage an den „Java Naming and Directory Service“ ermittelte Ergebnis wird vom „JNDI Akteur“ in Form eines Java-Objektes an den „Session Bean Nutzer“ übergeben. Bei den als Ergebnis von Anfragen an den „Java Naming and Directory Service“ gelieferten Objekten handelt es sich nicht immer um Objekte, die als „Daten“ zu betrachten sind. Vielmehr bietet der „Java Naming and Directory Service“ spezielle Unterstützung für „Proxy-Objekte“ wie „EJB Home-Objekte“ oder „EJB Proxy-Objekte“. Sie können ebenfalls im „Java Naming and Directory Service“ registriert werden. Jedes Mal wenn ein Client ein solches „Proxy-Objekt“ beim „JNDI Akteur“ anfragt, erzeugt dieser eine Kopie des im „Java Naming and Directory Service“ registrierten „Proxy-Objektes“ und sorgt auch für den Verbindungsaufbau zwischen dem neu erzeugten „Proxy-Objekt“ und dem durch das „Proxy-Objekt“ vertretenen „Original-Objekt“.
Im Unterschied zur Erzeugung von „Session Beans“ wird die Zerstörung eines Session Beans nicht explizit durch den „Session Bean Nutzer“ angestoßen. Stattdessen werden „Session Beans“ automatisch zerstört, wenn das letzte auf dieses „Session Bean“ verweisende „Proxy-Objekt“ zerstört wurde.

Ingesamt besteht die Definition eines „Session Bean“ Typs aus vier Teilen:

- Home Interface
  Das Home Interface definiert die Schnittstelle zwischen „Session Bean Nutzer“ und „EJB Home-Objekt“ und wird in Form einer Java-Interface-Klasse definiert.
- Component Interface
  Das Component Interface definiert die Schnittstelle zwischen „Session Bean Nutzer“ und „EJB Proxy-Objekt“. Es wird ebenfalls in Form einer Java-Interface-Klasse definiert.
- Session Bean Implementation Class
  Die „Session Bean Implementation Class“ definiert den Typ der Objekte, die vom Container zur Repräsentation von „Session Beans“ innerhalb der Java VM benutzt werden.
- deployment descriptor

Der Anwendungsprogrammierer muss keine Implementierung für die „EJB Home-Objekte“ und „EJB Proxy-Objekte“ angeben. Stattdessen werden die zur Implementierung der Funktionalität dieser Akteure notwendigen Java Klassen automatisch generiert. Dies geschieht typischerweise bei der Installation des betreffenden „Session Bean“ Typs im Container.

Entity Beans

Wie bereits bei der Vorstellung des „Online Shop“ Beispiels erwähnt, dienen „Entity Beans“ zur Repräsentation von Datensätzen, die in einem Datenbanksystem abgelegt sind. Jedem Typ von Datensatz entspricht dabei ein „Entity Bean“ Typ. Die vom EJB Container angebotene Unterstützung ist speziell auf relationale Datenbanksysteme ausgerichtet. Der Umfang, der von einem einzelnen „Entity Bean“ repräsentierten Daten, entspricht typischerweise genau einer Zeile einer Datenbanktabelle. Die Existenz eines „Entity Beans“ ist untrennbar an die Existenz der durch das „Entity Bean“ repräsentierten Daten gekoppelt. Dementsprechend ist das Erzeugen/Zerstören eines „Entity Beans“ untrennbar mit der Erzeugung/Löschung eines Datensatzes im Datenbanksystem verbunden. Zur Identifikation eines „Entity Beans“ dient üblicherweise der „Primärschlüssel“ des entsprechenden Datensatzes im Datenbanksystem.
Bevor weitere Details der vom Container bereitgestellten Unterstützung für „Entity Beans“ vorgestellt werden, soll noch einmal kurz auf die generellen Unterschiede zwischen „Entity Beans“ und „Session Bean“ eingegangen werden. Ein „Session Bean“ ist normalerweise einer Benutzersitzung zugeordnet. Es wird automatisch vom EJB Container zerstört, wenn kein „EJB Proxy-Objekt“ mehr existiert, welches auf das betreffende „Session Bean“ verweist. Der J2EE Standard verbietet nebenläufige Aufrufe von „Session Bean“-Methoden. Ganz anders ist die Situation bei „Entity Beans“. Die Existenz der ihnen repräsentierten Daten ist nicht an eine bestimmte Benutzersitzung gekoppelt. Die Löschung eines „Entity Beans“ und der zugehörigen Daten geschieht nicht automatisch, sondern muss explizit in Auftrag gegeben werden. Außerdem wird der nebenläufige Zugriff auf die durch ein „Entity Bean“ repräsentierten Daten vom Container explizit, durch Bereitstellung eines entsprechenden Transaktionskonzepts, unterstützt.
Ähnlich wie bei „Session Beans“ besteht auch bei „Entity Beans“ das Problem, dass es im Allgemeinen nicht möglich ist, sämtliche „Entity Beans“ durch entsprechende Java-Objekte innerhalb der zum Betrieb des EJB-Containers genutzten Java VM zu repräsentieren. Dies würde nämlich letztlich bedeuten, dass der gesamte im Datenbanksystem abgelegte Datenbestand in den Java Heap geladen werden müsste. Aufgrund der beschränkten Größe des Java Heap kann stattdessen stets nur ein verhältnismäßig kleiner Teil der „Entity Beans“ durch entsprechende Objekte im Heap der Java VM repräsentiert werden. Im Endeffekt kann die Gesamtheit der im Java Heap repräsentierten „Entity Beans“ als ein vom EJB Container verwalteter Datenbankpuffer betrachtet werden. Dabei trifft der Container die Entscheidung, welche „Entity Beans“ aktuell in Form entsprechender Objekte im Heap der Java VM repräsentiert werden und welche

nicht.
Analog zu der bereits vorgestellten Unterstützung für den Fernzugriff auf „Session Beans“ unterstützen J2EE Server auch den Fernzugriff auf „Entity Beans“. Der Fernzugriff auf „Entity Beans“ basiert ebenfalls auf der Verwendung entsprechender „EJB Proxy-Objekte“ und „EJB Home-Objekte“ innerhalb der betreffenden „Client Java VM“. Um diese Stellvertreter-Objekte für beliebige Komponenten eines Anwendungsprogramms zugänglich zu machen, können sie ebenso wie die Stellvertreter-Objekte für „Session Beans“ im „Java Naming and Directory Service“ registriert werden
Die Definition eines „Entity Bean“ Typs besteht genau wie die Definition eines „Session Bean“ Typs aus einem „Home Interface“, einem „Component Interface“, einer „Bean Implementation Class“ und einem „deployment descriptor“. Trotz dieser formalen Ähnlichkeiten gibt es doch wesentliche Unterschiede zwischen den beiden EJB Arten. Während die Methoden von „Session Beans“ primär der Codierung der „Business-Logik“ dienen, dient die in den Methoden von „Entity Bean“ Klassen codierte Funktionalität im Wesentlichen nur der Synchronisation der Attribute von „Entity Bean“-Objekten mit den zugehörigen Datensätzen im Datenbanksystem. Da in den „Entity Bean“ Methoden normalerweise keine „Business-Logik“ codiert ist, besteht das „Component Interface“ eines „Entity Beans“ üblicherweise nur aus Methoden die den Zugriff auf die einzelnen Attributwerte des „Entity Bean“ Objekts erlauben. Im Unterschied zum „Home Interface“ von „Session Beans“, welches typischerweise nur Methoden zum Anlegen neuer Beans umfasst, sind im „Home Interface“ eines „Entity Bean“ Typs üblicherweise auch Methoden definiert, um nach bestimmten, bereits vorhandenen Beans zu suchen und um Beans zu zerstören.
Die Abbildung zwischen den Attributen der Objekte der „Bean Implementation Class“ und den von einem „Entity Bean“ repräsentierten Datenbankinhalten folgt häufig dem Schema, dass ein Bean genau eine Zeile einer bestimmten Datenbanktabelle repräsentiert und dass in der „Bean Implementation Class“ jeweils ein Attribut für jede Tabellenspalte definiert ist. Da dieser Fall in der Praxis sehr häufig vorkommt, bietet der EJB Container hierfür besondere Unterstützung an, welche in der Literatur unter dem Schlagwort „Container Managed Persistence“ bekannt ist. Wenn eine derartige Abbildung zwischen Datenbankinhalten und Bean Attributen gewünscht wird, kann der Anwendungsprogrammierer die Zuordnung zwischen den in der „Bean Implementation Class“ definierten Objektattributen und den Tabellenspalten nämlich deklarativ durch entsprechende Einträge im „deployment descriptor“ festlegen. Macht er von dieser Möglichkeit Gebrauch, kann er auf die Implementierung der Methoden zur Synchronisation der Bean Attribute mit dem Inhalt der Datenbank verzichten. Stattdessen wird der hierfür notwendige Java/SQL-Code dann bei der Installation des Bean-Typs vom Container generiert.

Message Driven Beans

Im Unterschied zu den beiden bereits vorgestellten Arten von Enterprise Java Beans finden Message Driven Beans bei der Realisierung des „Online Shop“ Beispiels keine Verwendung. Message Driven Beans wurden zur Realisierung von Systemkomponenten konzipiert, deren Zweck in der Verarbeitung von Nachrichten besteht. Die Übergabe der zu verarbeitenden Nachrichten an das Message Driven Bean erfolgt dabei derart, dass sie in einer von dem Bean überwachten Message Queue hinterlegt werden. In Bild 28 ist diese Grundidee veranschaulicht.

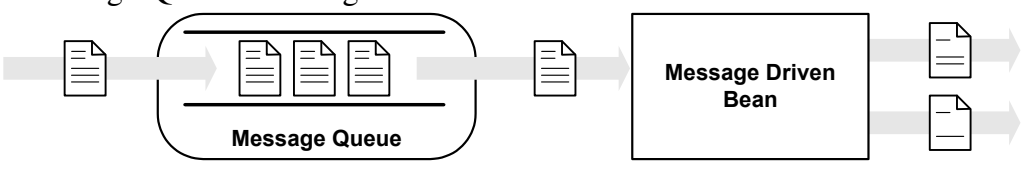


Bild 28: Message Driven Beans - Konzept

Durch die Nutzung einer Message Queue zur Übergabe der zu verarbeitenden Nachrichten wird der Prozess der Anlieferung von Nachrichten vom Prozess der Verarbeitung dieser Nachrichten entkoppelt.
Aufgrund der gerade vorgestellten Konzeption der Message Driven Beans steht ihre Verwendung immer mit der Nutzung entsprechender „Message Queues“ in Zusammenhang. Diese „Message Queues“ werden dabei nicht vom „EJB Container“ selbst, sondern von einem so genannten „Java Message Server“ verwaltet, dessen Funktionsweise ebenfalls durch die J2EE Spezifikation festgelegt ist. Daraus ergibt sich der in Bild 29 dargestellte, für die Nutzung von

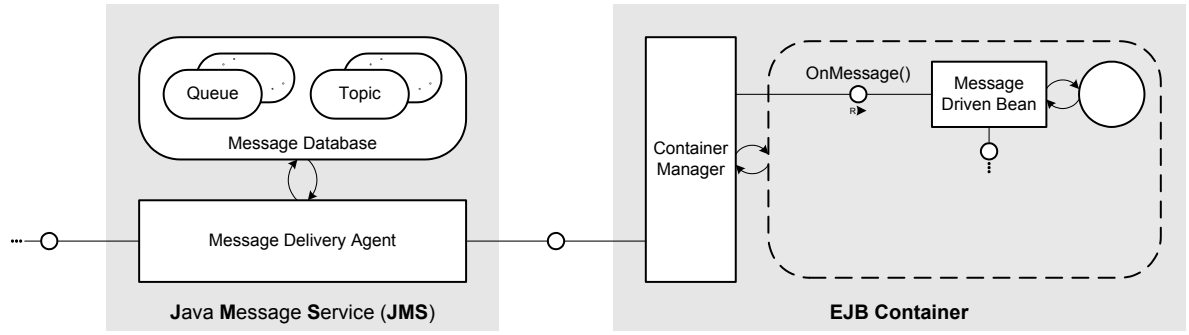


Bild 29: Verarbeitung von JMS-Messages durch Message Driven Beans

„Message Driven Beans“ charakteristische Systemaufbau. Die linke Bildhälfte zeigt den „Java Message Service“, während in der rechten Bildhälfte der EJB Container sowie die im Container beherbergten Message Driven Beans angedeutet sind. Der im Innern des „Java Message Service“ dargestellte, mit „Message Database“ bezeichnete Speicher dient als Nachrichtenpuffer. Alle eintreffenden Nachrichten werden zunächst in diesem Puffer abgelegt und anschließend an die entsprechenden Empfänger weitergegeben. Insgesamt unterstützt der Java Message Service zwei Arten der Nachrichtenweitergabe. Dies äußert sich im Bild darin, dass der „Message Database“ Speicher in Speicher für „Queues“ und „Topics“ unterteilt ist. Der Nachrichtenverbreitung über „Queues“ liegt die Vorstellung zugrunde, dass jede in einer „Queue“ hinterlegte Nachricht an genau einen Empfänger weitergeleitet wird. Hingegen hat man bei der Nachrichtenverbreitung über „Topics“ die Vorstellung, dass die eintreffenden Nachrichten an eine Liste von Abonnenten verteilt werden, wobei jeder Abonnent eine Kopie aller Nachrichten zu diesem „Topic“ erhält. Welche Nachrichten durch ein Message Driven Bean verarbeitet werden, wird bei der „Installation“ des Message Driven Beans im EJB Container festgelegt. Ein Message Driven Bean kann dabei sowohl als Abonnent eines „Topics“ auftreten als auch zur Verarbeitung von Nachrichten aus einer „Queue“ eingesetzt werden. Insgesamt wurde bei der Spezifikation des „Java Message Service“ großer Wert auf die Zuverlässigkeit des Nachrichtenaustauschs gelegt. Zur Realisierung des im Innern des „Java Message Service“ dargestellten „Message Database“ Speichers wird üblicherweise ein relationales Datenbanksystem eingesetzt. Um einen „sicheren“ Austausch von Nachrichten zu ermöglichen, geschieht der Austausch von Nachrichten unter Verwendung eines Zwei-Phasen-Commit Protokolls.

Jedes Message Driven Bean ist durch eine Java Klasse implementiert, in der eine „OnMessage()“-Methode definiert ist, welche den Ablauf zur Verarbeitung einer einzelnen Nachricht festlegt. Diese Methode wird vom EJB Container für jede eintreffende Nachricht einmal aufgerufen. Ebenso wie bei den bereits vorgestellten Arten von EJB’s ist der EJB Container auch bei Message Driven Beans für die Erzeugung / Freigabe der entsprechenden Bean Objekte zuständig und verwaltet einen Pool von Threads, die zur Ausführung der „On Message()“ Methoden der Message Driven Beans eingesetzt werden.

Obwohl Message Driven Beans bei der Realisierung des „Online Shop“ Beispiels keine Anwendung fanden, kommen sie dennoch im Kontext des Pet-Store Systems zur Anwendung. Sie werden dort zur Realisierung von „Workflow“-Funktionalität eingesetzt. Diese „Workflow“-Funktionalität dient dabei der automatisierten Bearbeitung der von den Kunden der Tierhandlung in Auftrag gegebenen Bestellungen. Insgesamt ist diese Funktionalität mit Hilfe von sieben Message Driven Beans, fünf Queues sowie einem Topic realisiert. Die Zusammenhänge zwischen diesen verschiedenen Komponenten sowie deren Umgebung sind in Bild 30 veranschau-

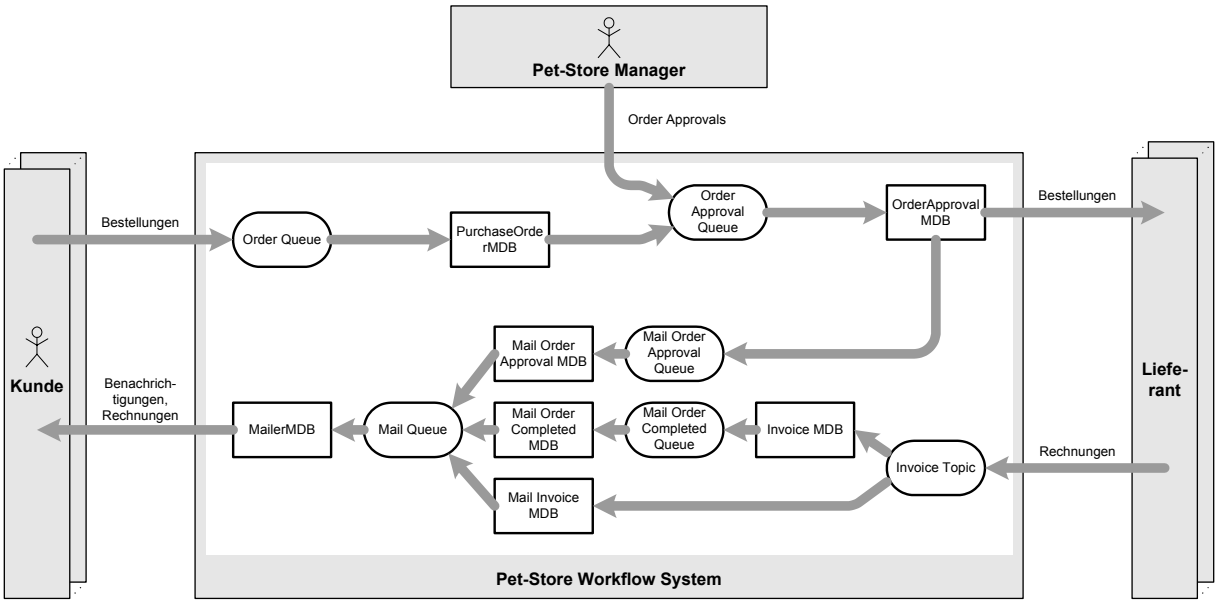


Bild 30: Bearbeitung von Bestellungen durch das Pet-Store System

licht. Dabei sind die Message Driven Beans durch die im Innern des „Pet-Store Workflow Systems“ dargestellten Rechtecke symbolisiert, während die Queues und der Topic durch ovale Knoten repräsentiert sind. Die zwischen den verschiedenen Komponenten ausgetauschten Messages „fließen“ dabei entlang der dargestellten Kanten, wobei die Pfeilrichtung die Richtung des Nachrichtenflusses angibt. Die Umgebung des „Pet-Store Workflow Systems“ besteht aus den Kunden, den Lieferanten und dem „Pet-Store Manager“.
An dieser Stelle sei angemerkt, dass Bild 30 kein FMC-Aufbaubild ist und dass im Rahmen der Verarbeitung einer Nachricht durch ein Message Driven Bean nahezu beliebige Berechnungen angestellt werden können. Insbesondere können dabei auch Session und Entity Beans „verwendet“ und die in der Pet-Store Datenbank abgelegten Unternehmensdaten verändert werden.

Der Prozess der Bearbeitung einer Bestellung wird durch die Übermittlung der Bestellung an die Tierhandlung angestoßen. Die Übermittlung der Bestellungen geschieht im vorliegenden Fall mit Hilfe der „Online Shop“ Anwendung, welche die Bestellungen in der mit „Order Queue“ bezeichneten Message Queue hinterlegt. Sämtliche dort hinterlegten Nachrichten werden von dem als „Purchase Order MDB“ bezeichneten Message Driven Bean zunächst einmal in die Pet-Store Datenbank eingetragen. Wenn der Wert der bestellten Waren einen bestimmten Schwellwert nicht überschreitet, wird die weitere Bearbeitung der Bestellung automatisch in die Wege geleitet, indem eine entsprechende Nachricht in der „Order Approval Queue“ hinterlegt wird. Überschreitet der Warenwert den Schwellwert, so muss die Bestellung vor der weiteren Bearbeitung zunächst durch den „Pet-Store Manager“ genehmigt werden. Hierzu benutzt dieser eine spezielle „Manager Anwendung“, welche für jede genehmigte Bestellung ebenfalls eine entsprechende Message in der „Order Approval Queue“ hinterlegt. Die in die „Order Approval Queue“ eingespeisten Genehmigungen werden vom „Order Approval MDB“ bearbeitet. Im

Rahmen der Bearbeitung dieser Messages werden einerseits entsprechende Bestellungen an die Lieferanten der Tierhandlung abgeschickt. Andererseits wird das Absenden dieser Bestellungen in der Pet-Store Datenbank vermerkt und der Versand einer Auftragsbestätigung an den Kunden in die Wege geleitet, indem eine Message in der „Mail Order Approval Queue“ hinterlegt wird. Die Auftragsbestätigung wird in der Folge vom „Mail Order Approval MDB“ erzeugt und in die „Mail Queue“ eingespeist. Die in der „Mail Queue“ hinterlegten Nachrichten werden vom „Mailer MDB“ verarbeitet. Jede dieser Nachrichten löst den Versand einer entsprechenden Email aus.
Aufgrund der an die Lieferanten erteilten Bestellungen liefern diese die bestellten Tiere im Namen der Tierhandlung an die Kunden aus. Nach erfolgter Lieferung stellen die Lieferanten diesbezügliche Rechnungen an die Tierhandlung. Damit die von den Lieferanten ausgestellten Rechnungen vom „Pet-Store Workflow System“ automatisch weiterverarbeitet werden können, werden sie über das „Invoice Topic“ in das System eingespeist. Dass die Lieferantenrechnungen nicht über eine Queue, sondern über ein Topic eingespeist werden, hat dabei den Grund, dass jede dieser Lieferantenrechnungen sowohl durch das „Invoice MDB“ als auch durch das „Mail Invoice MDB“ verarbeitet werden muss. Die durch das „Mail Invoice MDB“ geleistete Verarbeitung bewirkt, dass die Kunden Rechnungen erhalten. Dabei wird für jede von einem Lieferanten durchgeführte Teillieferung eine separate Rechnung an den Kunden versandt. Hingegen dient die Verarbeitung durch das „Invoice MDB“ dazu, die Teillieferungen in der Pet-Store Datenbank zu vermerken und die vollständige Erfüllung der Kundenbestellung festzustellen. Wenn die letzte Teillieferung einer Kundenbestellung erfolgt ist, wird der Versand einer entsprechenden Benachrichtigung an den Kunden in die Wege geleitet, indem eine Message in der „Mail Order Completed Queue“ hinterlegt wird.

3.3.2 Abbildung von J2EE Komponenten auf reale Maschinen

Die Komponenten eines J2EE Servers können auf unterschiedliche Weise mittels Betriebssystemprozessen beziehungsweise realen Maschinen realisiert werden. Im Extremfall werden sämtliche Komponenten eines J2EE Servers auf Basis einer einzigen Java VM realisiert. Eine andere, realistischere Form der Abbildung von J2EE Komponenten auf reale Maschinen ist in Bild 31 veranschaulicht. Die Darstellung geht von einer Verteilung der Komponenten auf drei verschiedene Rechner aus. Rechner 1 dient zur Realisierung des Java Message Service und wird außerdem zum Betrieb des hierfür notwendigen Datenbanksystems eingesetzt. Die beiden anderen Rechner dienen jeweils als Trägersystem für eine einzige Java VM. Die auf Rechner 2 betriebene Java VM dient als Trägersystem für den Web Container und die auf Rechner 3 betriebene Java VM dient als Trägersystem für den EJB Container. Die auf den Rechnern 2 und 3 vorhandenen Java VMs werden nicht ausschließlich zum Betrieb der Container, sondern außerdem auch für die als „Services (part)“ bezeichneten Komponenten benutzt. Die „Services (part)“ Komponenten wurden stellvertretend für diejenige Application Server Funktionalität eingezeichnet, die nicht unmittelbar einem bestimmten Container zuzurechnen ist. Zu dieser Funktionalität gehört insbesondere der bereits angesprochene „Java Naming and Directory Service“ sowie der „Deployment Service“. Der „Deployment Service“ dient dabei zur Installation und Inbetriebnahme von Anwendungen auf dem J2EE Server. Die in Bild 31 dargestellte Form der Verteilung der J2EE Komponenten auf verschiedene Rechner steht im Einklang mit der J2EE Spezifikation und kann mit Hilfe der verfügbaren J2EE Server Implementierungen auch ohne weiteres umgesetzt werden. Das genau dieses Szenario ausgewählt wurde, ist eher willkürlich. Es soll vielmehr als Grundlage dienen, um auf typische Probleme hinzuweisen, die bei der Verteilung von J2EE Komponenten auf verschiedene Rechner beziehungsweise auf verschiedene Java VM’s entstehen.

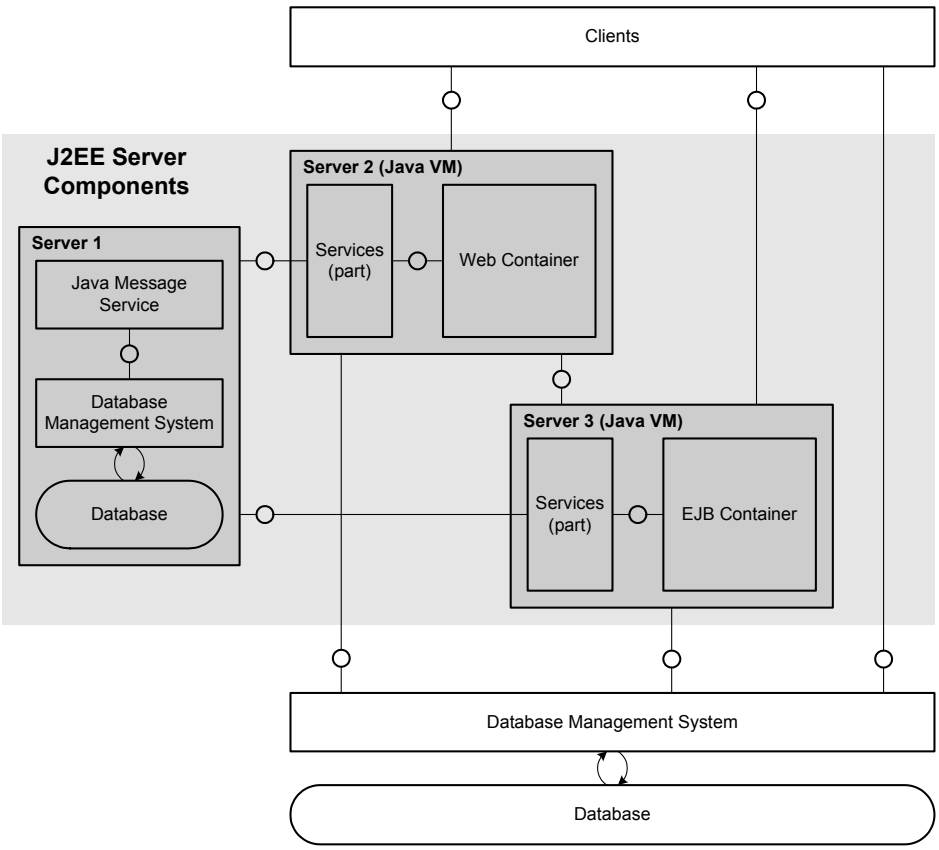


Bild 31: Mögliche Realisierung eines J2EE Servers unter Verwendung von drei Rechnern

- Verteilung des Web Containers und des EJB Containers
  Die J2EE Spezifikation wurde so gestaltet, dass eine Verteilung von Web Container und EJB Container auf unterschiedliche Rechner möglich ist. Aus diesem Grund wurde die dargestellte Form der Verteilung letztlich auch als „Referenz“ für die Diskussion der Verteilungsproblematik ausgewählt. Wenn die beiden Container eines J2EE Servers tatsächlich auf unterschiedliche Rechner verteilt werden, so ist dies im Allgemeinen mit einer wesentlich höheren Latenzzeit bei der Kommunikation zwischen solchen in unterschiedlichen Containern beherbergten Anwendungskomponenten verbunden. Um einen wirklichen Nutzen durch diese Form der Verteilung zu erzielen, ist es daher erforderlich, dies bereits bei der Entwicklung der betreffenden Anwendungen zu berücksichtigen.
- Skalierbarkeit
  Legt man die Anzahl an Nutzern zugrunde, die mit Hilfe eines SAP R/3 Systems sinnvoll bedient werden können, so ist es unrealistisch anzunehmen, dass vergleichbare Systeme unter Verwendung von nur drei Rechnern realisiert werden können. Die verteilte Realisierung eines Web Containers oder eines EJB Containers auf Basis mehrerer Java VM’s ist jedoch in der J2EE Spezifikation nicht vorgesehen. Die einzige Möglichkeit, einen größeren Benutzerkreis mit Hilfe eines auf J2EE Technologie basierenden Systems zu bedienen, besteht daher in der Verwendung mehrerer Web Container und EJB Container. Eine solche Form der Verteilung bringt jedoch erheblichen Administrationsaufwand mit sich. Außerdem muss auch diese Art der Verteilung im Allgemeinen ebenfalls bei der Entwicklung der betreffenden Anwendungsprogrammkomponenten berücksichtigt werden.
- Abschottung von Sitzungen verschiedener Benutzer
  Ein grundsätzliches Problem bei J2EE Servern ist durch die Verwendung von Java VM’s als Trägersystem für die Komponenten des Application Servers gegeben. Die Java VM

stellt kein Abschottungskonzept zur Verfügung, das mit dem aus der Betriebssystemwelt bekannten Prozess-Konzept vergleichbar ist. In Bezug auf die Implementierung eines J2EE Servers hat dies zur Folge, dass es unmöglich ist, den Bedarf an Betriebsmitteln zu überwachen, der für die Bereitstellung der einzelnen EJB Side User Agents und Web Side User Agents aufgewendet wird. Auch wenn dies möglich wäre, bestünde praktisch keine Möglichkeit, die Betriebsmittel, die zur Bereitstellung eines „außer Kontrolle geratenen" User Agents reserviert wurden, wieder einer sinnvollen Nutzung zuzuführen. Potentiell können daher Fehler, die bei der Bearbeitung eines Requests auftreten, dessen Bearbeitung einem einzelnen EJB Side User Agent oder Web Side User Agent zuzurechnen ist, zur Beeinträchtigung der gesamten Java VM führen, die den betreffenden User Agent beherbergt. Die Konsequenzen solcher Fehler sind daher keineswegs auf die Sitzung des Benutzers beschränkt, bei dem der Fehler in Erscheinung getreten ist, sondern wirken sich möglicherweise auf die Sitzungen aller Nutzer des J2EE Servers aus. Legt man beispielsweise das in Bild 31 dargestellte Verteilungsszenario zugrunde, so würde ein Absturz der Java VM, die der Realisierung des EJB Containers dient, praktisch den gesamten J2EE Server lahm legen. Insbesondere würden in einem solchen Fall auch sämtliche Web Side User Agents nutzlos werden, die das Vorhandensein eines ihnen zugeordneten EJB Side User Agents voraussetzen.

- Java VM's und Garbage Collection
  Speziell in der Vergangenheit hat es sich als unzweckmäßig erwiesen, einen Rechner ausschließlich zum Betrieb einer einzigen Java VM zu benutzen, da auf diese Weise keine akzeptable Auslastung des betreffenden Rechners erzielt werden konnte. Obwohl bei der Implementierung von Java VM's inzwischen deutliche Fortschritte erzielt wurden, sind die durch die Garbage Collection verursachten „Betriebspausen" auch heute noch sehr problematisch. Daher neigt man auch heute noch dazu, auf Rechnern, die der Realisierung von J2EE Serverkomponenten dienen, mehrere Java VM's zu betreiben, denen jeweils nur ein Teil des realen Hauptspeichers zur Verfügung gestellt wird.

Aufgrund der genannten Probleme ist es für ein auf J2EE Technologie basierendes Informationssystem typisch, dass seine Realisierung auf eine Vielzahl von Java VM's verteilt ist. Die Hersteller von J2EE Server Implementierungen berücksichtigen dies beim Entwurf der verschiedenen Serverkomponenten und stellen darüber hinaus entsprechende Tools zur Verwaltung der verschiedenen Java VM's eines Application Server Clusters zur Verfügung. Anhand von Bild 32 soll nun kurz auf die Implementierung des In-Q-My[22] Application Servers eingegangen werden. Ein unter Verwendung der In-Q-My Implementierung hergestellter Java VM Cluster besteht aus einer variablen Anzahl von Java VM's, wobei jede dieser Java VM's entweder zur Realisierung eines Dispatchers oder zur Realisierung eines App Servers verwendet wird. Die verschiedenen Dispatcher und App Server können auf vielfältige Art auf reale Rechner verteilt werden. Jeder der App Server realisiert einen nahezu vollständigen J2EE Server, er beinhaltet einen eigenen Web Container, EJB Container sowie eine Service Komponente. Die App Server stehen nicht in direktem Kontakt mit den Clients. Stattdessen nehmen die Dispatcher die von den Clients eintreffenden Requests entgegen und teilen sie den App Servern zur Bearbeitung zu. Zur Kommunikation innerhalb des Server Clusters wird dabei ein In-Q-My spezifisches Kommunikationsprotokoll verwendet. Die Dispatcher haben daher nicht nur die Aufgabe, den für die Bearbeitung eines Requests zuständigen App Server auszuwählen, sondern sind außerdem dafür zuständig, die verschiedenen Protokolle, die die Clients zur Kommunikation mit

22. Die Rechte am In-Q-My Application Server sind inzwischen von der SAP AG aufgekauft worden. Die Entwicklung des Servers wurde jedoch fortgesetzt. Die Nachfolgeprodukte werden derzeit als Bestandteil des SAP Web Application Servers vermarktet.

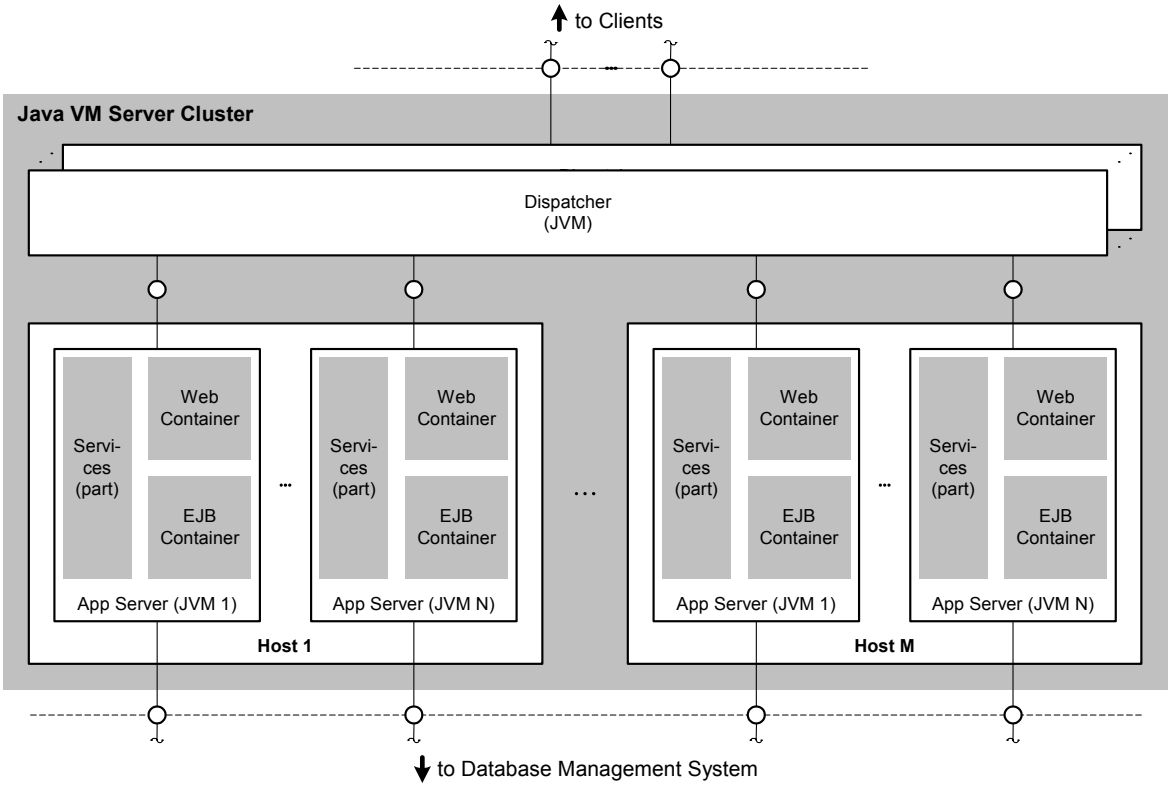


Bild 32: J2EE Server Cluster

den J2EE Server Komponenten verwenden können, in das für die clusterinterne Kommunikation verwendete Protokoll „umzuwandeln“. Speziell, wenn die Kommunikation zwischen Clients und Server verschlüsselt erfolgt, ergibt sich hierdurch der Vorteil, dass die App Server von der rechenintensiven Aufgabe der Ver-/Entschlüsselung der Kommunikationspakete entlastet werden. Die In-Q-My Implementierung unterstützt unterschiedliche Modelle zur Verteilung von Anwendungsprogrammkomponenten auf die verschiedenen App Server, die zu einem In-Q-My Cluster gehören. Wenn auf dem J2EE Server Cluster mehrere voneinander unabhängige J2EE Anwendungen betrieben werden, so werden die verschiedenen Anwendungen typischerweise auf eigens dafür bereitgestellten App Servern installiert. Eine derartige Verteilung käme der Verwendung unterschiedlicher J2EE Server für die unterschiedlichen Anwendungsprogramme gleich und ist daher stets problemlos möglich. Die eigentliche Stärke solcher Application Server Cluster Realisierungen liegt aber darin, das gleiche Anwendungsprogramm auf mehreren App Servern zu installieren, so dass eine höhere Zahl an Benutzern zeitgleich bedient werden kann. Wie bereits angesprochen, ist eine derartige Nutzung von J2EE Anwendungsprogrammen nur dann sinnvoll möglich, wenn dies bereits bei der Erstellung der betreffenden Anwendungsprogramme berücksichtigt wurde. Die Unterstützung für diese Art der Verteilung ist beim In-Q-My Server insbesondere durch zwei Dinge gegeben: Einerseits werden die Dispatcher ständig über den Belastungszustand der verschiedenen zum Server Cluster gehörenden App Server informiert und verfügen über die Fähigkeit, diese Information bei der Zuteilung von eintreffenden Requests zu App Servern derart zu berücksichtigen, dass eine Lastverteilung zwischen denjenigen App Servern erzielt wird, die zum Betrieb der gleichen Anwendungsprogramme benutzt werden. Andererseits ist diese Form der Verteilung auch bei der Implementierung der J2EE Server Services berücksichtigt worden.

3.4 Zweck der Integration von J2EE Technologie in den SAP Application Server

Die SAP AG hat vor über 15 Jahren mit der Entwicklung des als Basis für das R/3 System dienenden Application Servers begonnen. Heute verfügt sie über eine weltweit, mit sehr großem Erfolg eingesetzte Application Server Technologie. Die auf der Java Plattform basierende J2EE Technologie ist hingegen viel jünger und ist in wesentlichen Punkten deutlich unausgereifter als die derzeit verfügbare, ABAP basierte Lösung. Dennoch ist die Unterstützung der J2EE Technologie aus verschiedenen Gründen von Bedeutung für die SAP:

- Integrationsbedarf
  Die J2EE Technologie erfreut sich in letzter Zeit zunehmend größerer Beliebtheit und es werden inzwischen eine Vielzahl von Systemen angeboten, die auf der J2EE Technologie basieren. Der Einsatz solcher Systeme erfordert jedoch oft eine Kopplung an bereits vorhandene Informationssysteme. Daraus ergibt sich ein starkes Interesse der SAP Kunden an einer leistungsfähigen Integration von bestehenden R/3 Systemen und solchen, die auf J2EE Technologie basieren.
- Bedarf nach leistungsfähigeren J2EE Application Servern
  Obwohl bereits eine Vielzahl von J2EE Application Server Implementierungen auf dem Markt sind, weisen die auf Basis dieser Implementierungen erstellten Systeme auch heute noch Defizite auf. Diese Defizite betreffen insbesondere die maximale Anzahl der gleichzeitig unterstützten Benutzer sowie die Betriebssicherheit. Hier besteht für die SAP die Chance, ihre bestehende Kompetenz in der Konstruktion von Application Servern zu nutzen, um auch im Bereich der J2EE Application Servertechnologie zum Marktführer zu werden.
- Kundenwunsch
  Die Programmiersprache ABAP ist eine „SAP Spezialsprache“ und muss von Einsteigern zunächst erlernt werden, bevor sie mit der Entwicklung von Anwendungsprogrammen beginnen können. Aufgrund des hohen Verbreitungsgrades der Programmiersprache Java lassen sich verhältnismäßig leicht Programmierer finden, die die Sprache Java bereits beherrschen. Da insbesondere auch die SAP Kunden oft einen beträchtlichen Entwicklungsaufwand zur Anpassung des erworbenen Systems betreiben, drängen sie auf eine stärkere Unterstützung der J2EE Technologie durch die SAP. Sie versprechen sich davon einerseits eine Senkung der Entwicklungskosten, da Java Entwickler kostengünstiger sind. Andererseits sehen sie in der Verwendung der J2EE Technologie die Möglichkeit, eine größere Unabhängigkeit vom SAP Application Server zu erreichen.
- ABAP und der Stand der Technik
  Seit dem ursprünglichen Design der Programmiersprache ABAP haben sich viele Dinge verändert, die auf dieses ursprüngliche Design Einfluss genommen haben. Einerseits sind dies Neuerungen auf dem Gebiet der Programmiersprachen. Beispielsweise gab es das Konzept der Objektorientierung zum Zeitpunkt der Erfindung von ABAP noch nicht. Andererseits haben sich auch die Anforderungen der Kunden an Informationssysteme geändert. Dies betrifft insbesondere die Möglichkeiten zur Vernetzung solcher Informationssysteme wie auch die kundenspezifische Anpassung solcher Systeme. Beispielsweise erweisen sich heute Anwendungsprogramme, die von einem bestimmten, „in das Anwendungsprogramm hineinprogrammierten“ Dialoggraphen ausgehen, vielfach als zu unflexibel für eine kundenspezifische Anpassung. Diese Veränderungen wurden natürlich bei der Weiterentwicklung von ABAP berücksichtigt, jedoch mussten bei der Weiterentwicklung von ABAP aus Gründen der Abwärtskompatibilität vielfach Kompromisse eingegangen

werden. Es wird daher darüber nachgedacht, die Weiterentwicklung der J2EE Technologie über die bloße Unterstützung hinaus aktiv voranzutreiben und langfristig auf Java/ J2EE Technologie umzusteigen.

Derzeit wird die Verwendung von J2EE Technologie bereits durch den SAP Web Application Server unterstützt. Bild 33 gibt einen Überblick über die Komponenten, die auf einem Rechner

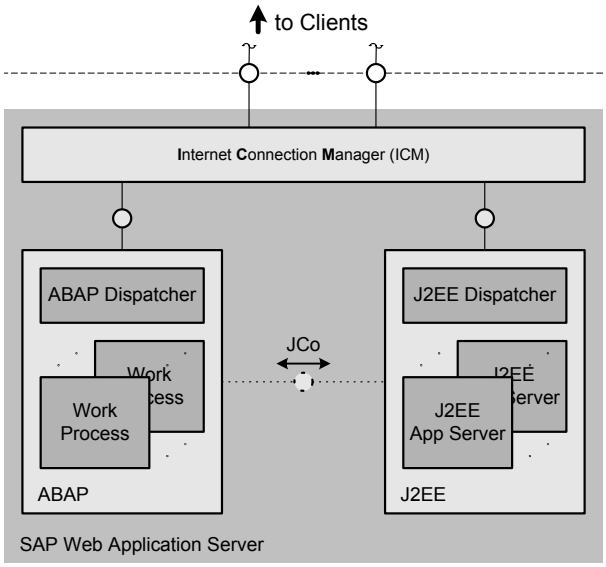


Bild 33: SAP Web Application Server

vorzufinden sind, der Bestandteil eines SAP Web Application Server Clusters ist. Im Wesentlichen handelt es sich bei einem solchen Rechner um einen konventionellen, ABAP basierten Application Server, auf dem zusätzlich ein spezieller J2EE Server betrieben wird. Die Kopplung zwischen ABAP und J2EE Server ist mit Hilfe eines zur „Java Connector Architecture“ konformen Connectors realisiert, der als „JCo“ bezeichnet wird. Der ABAP und J2EE Serverteil sind über den **I**nternet **C**onnection **M**anager (ICM) mit der Außenwelt verbunden. Für die Kommunikation über die Kanäle zwischen ICM, dem ABAP Server und dem J2EE Server wird eine speziell auf diesen Fall optimierte, auf „shared memory“ basierende Implementierung bereitgestellt.
Im Vergleich zur Verwendung unterschiedlicher Rechner für den ABAP und den J2EE Server gestattet der dargestellte Systemaufbau eine effizientere Bearbeitung solcher Requests, zu deren Bearbeitung die Auswertung von ABAP und Java Anwendungsprogrammteilen erforderlich ist. Dass im Rahmen eines Requests sowohl ABAP als auch Java Programmteile zur Ausführung kommen, tritt beispielsweise dann auf, wenn ein Benutzer eine J2EE Anwendung nutzt, die auf solche in ABAP definierte Funktionalität zurückgreift. In solchen Fällen ist dem betreffenden Benutzer sowohl eine ABAP-Sitzung als auch eine J2EE-Sitzung zugeordnet. Nun beruht die angesprochene Effizienzsteigerung darauf, dass bei der Kommunikation zwischen dem ABAP Server und dem J2EE Server der effizientere, „shared memory“ basierte Kommunikationspfad genutzt wird. Dies ist natürlich nur dann möglich, wenn die ABAP-Sitzung und die J2EE-Sitzung des betreffenden Benutzers auf dem gleichen Rechner beherbergt sind. Damit die ABAP-Sitzung und die J2EE-Sitzung eines Benutzers auch tatsächlich immer auf demselben Rechner beherbergt werden, wurde das beim Sessionaufbau zur Anwendung kommende Rechner-Session-Zuordnungsverfahren entsprechend angepasst.
Obwohl durch die Verwendung des Web Application Servers eine verbesserte Kopplung zwischen der ABAP und der J2EE Seite erreicht wird, ist der Aufwand für VM übergreifende Auf-

rufe immer noch um mehrere Größenordnungen höher als für normale Funktions-/ Methodenaufrufe“.

Mit der durch den SAP Web Application Server erzielten Verbesserung der Kopplung zwischen der ABAP und der J2EE Seite können die zu Beginn dieses Abschnitts erwähnten Unternehmensziele daher nur teilweise erreicht werden. Aus diesem Grund wird innerhalb der SAP weiterhin nach Möglichkeiten zur Verbesserung der Integration der ABAP und der J2EE Welt gesucht. Das in dem nun folgenden Abschnitt beschriebene VM Container Projekt ist ein solches Projekt, das der Suche nach solchen Möglichkeiten dient.

3.5 „VM Container“ - Projekt

Den Ausgangspunkt für das in diesem Kapitel beschriebene Projekt zur Integration einer Java VM in den SAP Application Server bildeten die im vorangegangenen Abschnitt 3.4 geschilderten Unternehmensziele. Der Grundgedanke des Projekts bestand darin, möglichst viele bewährte Konzepte der klassischen ABAP Application Server Technologie auf die J2EE Technologie zu übertragen und gleichzeitig eine größtmögliche Kompatibilität zum J2EE Standard zu wahren. Einer der wichtigsten Grundpfeiler für die Betriebssicherheit und Skalierbarkeit der klassischen Application Server Technologie liegt in der starken Abschottung von Sitzungen verschiedener Benutzer durch das bereits erwähnte Work Prozess Konzept. Da die J2EE Technologie gerade in diesem Punkt Schwächen aufweist, erschien die Übertragung dieses Konzepts auf die J2EE Technologie als besonders gewinnbringend.

Wie bereits in Abschnitt 3.3.2 erwähnt wurde, ist eine brauchbare Abschottung der Sitzungen verschiedener Benutzer innerhalb eines J2EE Servers aufgrund der fehlenden Unterstützung durch die Java VM nicht möglich. Daher bestand die Grundidee zur Erzielung der wechselseitigen Abschottung von Sitzungen verschiedener Benutzer darin, jedem Benutzer einen separaten J2EE Server zur Verfügung zu stellen. Die Realisierung dieser benutzerspezifischen J2EE Server sollte dabei, dem bestehenden „Screen Prozessor / ABAP VM“ Beispiel folgend, auf Basis von Work Prozessen realisiert werden, die jeweils einen multiplexfähigen J2EE Server implementieren. Genauer gesagt war es das Ziel die Implementierung des bestehenden Application Servers so zu erweitern, dass die Work Prozesse neben einem multiplexfähigen Screen Prozessor und einer multiplexfähigen ABAP VM zusätzlich einen multiplexfähigen J2EE Server beherbergen. Bild 34 veranschaulicht diese Idee anhand eines Ausschnitts aus

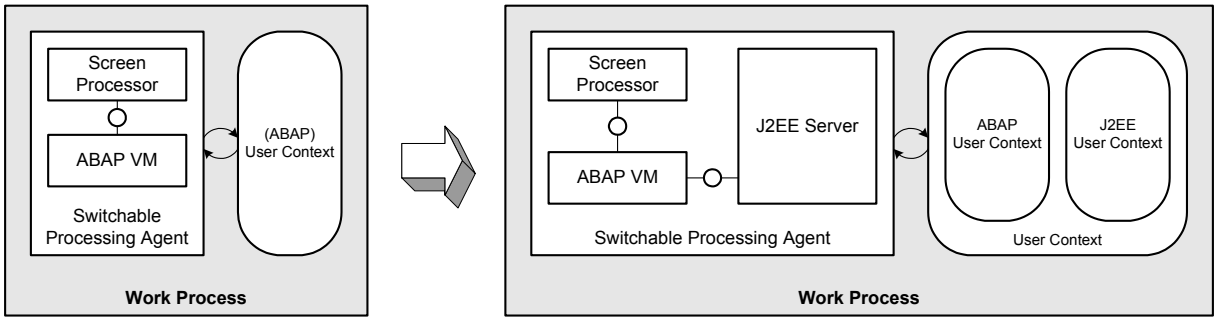


Bild 34: Integration eines multiplexfähigen J2EE Servers in den klassischen Application Server

Bild 21. Links ist der Aufbau eines klassischen Work Prozesses veranschaulicht, während in der rechten Bildhälfte der um die J2EE Funktionalität erweiterte Work Prozess veranschaulicht

wurde. Der rechts dargestellte Work Prozess unterscheidet sich von dem links dargestellten zum einen dadurch, dass der Switchable Processing Agent um die J2EE Server Funktionalität erweitert wurde. Zum anderen besteht der User Context nun aus zwei Teilen, einem ABAP und einem J2EE User Context. Der ABAP User Context enthält dabei die klassischen User Context Daten, während der J2EE User Context den Zustand des J2EE Servers beinhaltet. Im Vergleich zu dem im vorangegangenen Abschnitt 3.4 beschriebenen Web Application Server Ansatz weist der gerade beschriebene neben der starken Abschottung von Sitzungen verschiedener Benutzer noch eine Reihe weiterer Vorteile auf. Der größte dieser Vorteile besteht darin, dass er eine sehr leistungsfähige Kopplung zwischen der ABAP und der J2EE Welt zulässt, da sämtliche (ABAP, Web, EJB) Server Side User Agents per Konstruktion durch den gleichen Betriebssystemprozess realisiert sind. Außerdem sind eventuell auftretende Garbage Collection Pausen völlig unkritisch, da sie sich jeweils nur auf den J2EE Server eines Benutzers auswirken. Dadurch, dass jeder Benutzer über seinen eigenen J2EE Server verfügt, ergeben sich überdies hervorragende Voraussetzungen für das Debugging von J2EE Sessions.

Obwohl der vorgestellte Lösungsansatz große Vorteile mit sich bringt, war seine Umsetzung zunächst umstritten. Insbesondere war es unklar, ob die Entwicklung der dazu notwendigen Implementierung eines multiplexfähigen J2EE Servers mit vertretbarem Aufwand machbar wäre. Daher wurde das VM Container Projekt zunächst als Machbarkeitsstudie angelegt. Um die grundsätzliche Machbarkeit einer solchen Implementierung möglichst früh einschätzen zu können, wurden in Bezug auf den Funktionsumfang der im folgenden vorgestellten „prototypischen“ Implementierung einige Einschränkungen in Kauf genommen:
Bereits vor Beginn des Projekts war klar, dass der wesentliche Teil des Entwicklungsaufwandes in die Entwicklung einer für diesen Zweck geeigneten Java VM Implementierung fließen würde. Daher erschien es zweckmäßig, eine einfache, Interpreter basierte Java VM Implementierung als Ausgangsbasis für die Entwicklung dieser Implementierung zu wählen. Nach Evaluation verschiedener Java VM Implementierungen fiel die Wahl schließlich auf die von der Firma Sun hergestellte CVM Implementierung, die bereits in Abschnitt 2.2 näher vorgestellt wurde[23].
Eine weitere Einschränkung, die bei der Implementierung des Prototypen in Kauf genommen wurde, betrifft den Umfang der implementierten J2EE Funktionalität. Hier bestand das primäre Ziel in der Bereitstellung eines voll nutzbaren Web Containers, während die Unterstützung der EJB Container Funktionalität als zweitrangig angesehen wurde. Als Ausgangsbasis für die Implementierung der eigentlichen J2EE Funktionalität dienten die Referenz Implementierung des Web Containers [Tomcat] sowie die SAP eigene J2EE Server Implementierung (InQMy).

Im weiteren Verlauf dieses Abschnitts folgt nun eine detailliertere Darstellung der Implementierung des gerade vorgestellten Konzepts. Da es sich hierbei um eine Erweiterung des bestehenden Application Servers handelt, muss in Abschnitt 3.5.1 zunächst noch einmal etwas genauer auf die Implementierung des bestehenden Application Servers eingegangen werden. Anschließend werden die notwendigen Maßnahmen zur Integration der CVM erläutert. In Abschnitt 3.5.3 wird auf die Implementierung der eigentlichen J2EE Server Funktionalität eingegangen und die hierzu notwendigen Anpassungen an den verwendeten Web-/EJB-Container Implementierungen vorgestellt. Danach folgt eine Diskussion der Stärken und Schwächen des erreichten Integrationsergebnisses.

23. Diese Implementierung ist dem „**C**onnected **L**imited **D**evice **C**onfiguration (CLDC) Application Profile“ zugeordnet und wird daher standardmäßig mit „**J**ava **2** **M**icro **E**dition (J2ME)“ Class Libraries ausgeliefert. Zum Betrieb eines J2EE Servers sind jedoch „**J**ava **2** **S**tandard **E**dition (J2SE)“ Class libraries erforderlich. Aufgrund der in der CVM vorhandenen Unterstützung für das „**C** **N**ative **I**nterface (CNI)“ war der upgrade auf J2SE Class Libraries jedoch problemlos möglich.

3.5.1 VM Container Framework

Betrachtet man die Implementierung des klassischen SAP Application Servers, so lassen sich verschiedene klar gegeneinander abgegrenzte Implementierungsteile finden, die von verschiedenen Gruppen innerhalb der SAP entwickelt werden. Grob besteht die Implementierung des Application Servers aus drei Teilen: ABAP-VM (inklusive Screen Prozessor), Database Interface und VM Container. Die Implementierung des Work Prozess Konzepts gehört dabei zu dem als VM Container bezeichneten Teil. Da die in diesem Kapitel beschriebene Integration einer Java VM in den SAP Application Server das bestehende VM Container Framework als Ausgangsbasis nutzt, wird dieses Framework nun anhand von Bild 35 genauer vorgestellt.

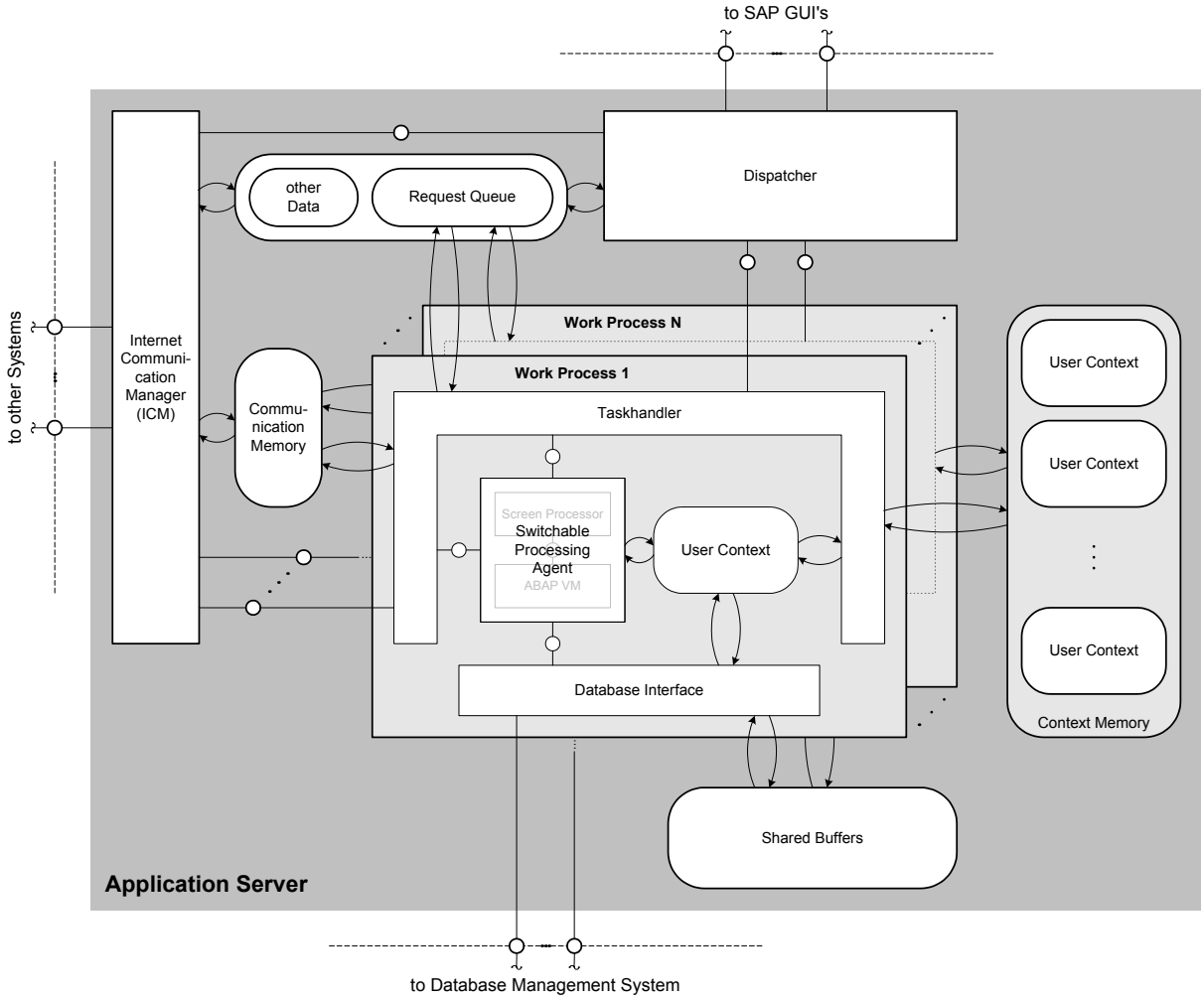


Bild 35: Application Server mit ICM

Viele der in Bild 35 dargestellten Komponenten kamen bereits in Bild 21 vor, welches der Veranschaulichung des Work Prozess Konzepts diente. Im Einzelnen sind dies: der Dispatcher, die Work Prozesse, die Request Queue, die Speicher für User Contexte sowie der mit Shared Buffers bezeichnete Speicher. Die Unterschiede zwischen den beiden Bildern sind durch zwei Dinge begründet. Einerseits wurde der innere Aufbau der Work Prozesse weiter verfeinert. Ein Workprozess besteht nun aus einem Taskhandler, einem Switchable Processing Agent, einem Database Interface Akteur sowie dem Speicher für den aktuell eingestellten User Context. Andererseits ist der Internet Communication Manager sowie das Communication Memory neu hinzugekommen. Der Internet Communication Manager ist dabei ebenso wie die Work Prozes-

se und der Dispatcher als eigener Betriebssystemprozess realisiert.
Wie bereits beschrieben, dient ein Work Prozess der Realisierung von Server Side User Agents. Dabei kann er nacheinander die Rollen verschiedener User Agents übernehmen. Die Umschaltung eines Work Prozesses zwischen den Rollen der verschiedenen Server Side User Agents geschieht durch Austausch des User Contextes. Dieser Austausch umfasst zwei Schritte, die im SAP Umfeld mit „Roll Out“ und „Roll In“ bezeichnet werden. Der Terminus „Roll Out“ steht dabei für das Entfernen des gerade aktiven User Contextes aus dem Work Prozess, während der Begriff „Roll In“ den Vorgang der Installation des neuen User Contextes im Work Prozess bezeichnet. Die Aufgabe der verschiedenen Akteure, die im Innern der Work Prozesse dargestellt sind, lassen sich am einfachsten anhand des Ablaufs zur Bearbeitung eines von einem SAP GUI initiierten Requests klären: Die von den SAP GUI’s kommenden Requestdaten werden vom Dispatcher entgegengenommen und in der Request Queue gepuffert. Wenn der Request vollständig beim Server eingetroffen ist, wird der Request einem gerade unbeschäftigten Work Prozess zur Bearbeitung zugeteilt. Hierzu besitzt der Dispatcher je einen Kanal zu den Taskhandlern der verschiedenen Workprozesse. Nach Erhalt eines zu bearbeitenden Requests überprüft der Taskhandler des betreffenden Workprozesses zunächst, ob der aktuell eingestellte User Context zu dem Request passt. Falls nicht, tauscht er den User Context gegen den passenden aus. Danach beauftragt er den Switchable Processing Agent mit der inhaltlichen Bearbeitung des Requests. Wenn die inhaltliche Bearbeitung des Requests durch den Switchable Processing Agent abgeschlossen ist, meldet er dies an den Dispatcher, der daraufhin die Antwortdaten an das betreffende SAP GUI versendet. Der Switchable Processing Agent ist für die Auswertung des Codes der Anwendungsprogramme zuständig. Im Fall eines klassischen SAP Application Servers besteht er daher aus einem Screen Prozessor sowie aus einer ABAP VM. Die Anwendungsdaten und -programme sind im klassischen Fall sämtlich in der zentralen Datenbank des Systems abgelegt. Zum Zugriff auf die zentrale Datenbank bedienen sich der Screen Prozessor und die ABAP VM des Datenbank Interface Akteurs. Der Database Interface Akteur ist in der Lage, die in der SAP eigenen Datenbankanfragesprache „Open SQL“ formulierten Anfragen zu bearbeiten und ist überdies für die Verwaltung des Datenbankpuffers zuständig, der im Bild in Form des „Shared Buffers“ Speicher repräsentiert ist.
Bei dem gerade beschriebenen Requestbearbeitungsablauf wurde davon ausgegangen, dass der zu bearbeitende Request von einem SAP GUI initiiert wurde und dass die Bearbeitung des Requests, abgesehen von den Zugriffen auf die zentrale Datenbank, keinerlei Kommunikation mit anderen Systemen erfordert. Aus diesem Grund war der im Bild dargestellte Internet Communication Manager nicht an der Requestbearbeitung beteiligt. Nun wurde bei der Vorstellung des SAP Web Application Servers jedoch bereits ein Fall erwähnt, auf den der gerade vorgestellte Ablauf zur Bearbeitung von Requests nicht zutrifft. Dabei handelt es sich um den Fall, dass bei der Bearbeitung eines Requests durch den J2EE Serverteil ein Request an den ABAP Serverteil gestellt wurde. Neben der Unterstützung für SAP GUI’s und J2EE Server als Requestquellen unterstützt der SAP Application Server noch diverse weitere Requestquellen, wobei zur Übermittlung von Requests an den Server verschiedene Protokolle verwendet werden können. Requests, zu deren Übermittlung nicht das SAP GUI Protokoll verwendet wird, werden nicht direkt vom Dispatcher entgegengenommen, sondern vom Internet Communication Manager. Neben der Aufgabe von außen an den Server gesandte Requests entgegenzunehmen, hat der Internet Communication Manager eine weitere Aufgabe, die mit ausgehenden (TCP-) Verbindungen in Zusammenhang steht. Unter ausgehenden Verbindungen sind dabei solche Verbindungen zu verstehen, deren Erzeugung während der Bearbeitung von Requests, durch entsprechende Anweisungen im Code von Anwendungsprogrammen explizit verlangt wird[24]. Dabei unterstützt der SAP Application Server auch solche Verbindungen, die über die Dauer eines Requests hinaus erhalten bleiben. Da diese Verbindungen auf Basis der Socket Implementierung des Betriebssystems realisiert werden, muss es einen Prozess geben, der gegenüber dem

Betriebssystem als Eigentümer des zur Realisierung der Verbindung dienenden Sockets auftritt. Ein potentieller Kandidat für diese Eigentümerschaft ist derjenige Workprozess, der den Request bearbeitet, in dessen Verlauf der Verbindungsaufbau verlangt wird. Dies wäre jedoch eine sehr unzweckmäßige Zuordnung, da der eigentliche Eigentümer der Verbindung der Server Side User Agent ist, dem der „Verbindungsaufbau-Request“ zuzurechnen ist. Die Unzweckmäßigkeit dieser Zuordnung liegt darin, dass alle nachfolgenden Requests, die dem betreffenden Server Side User Agent zuzurechnen sind, dann nur durch denjenigen Work Prozess bearbeitet werden könnten, der auch den „Verbindungsaufbau-Request“ bearbeitet hat. Um dieses Problem zu vermeiden und die freie Zuordenbarkeit von „User Contexten“ zu Workprozessen zu erhalten, tritt der Internet Communication Manager Prozess als Eigentümer solcher Sockets auf. Beim Datentransport über ausgehende Verbindungen kommt ihm daher die Aufgabe eines Vermittlers zwischen dem Betriebssystem und den Workprozessen zu. Der Datenaustausch zwischen dem Internet Communication Manager und den Work Prozessen geschieht primär über den im Bild als Communication Memory bezeichneten Speicher, der durch „shared memory“ realisiert ist. Darüber hinaus sind die Taskhandler der verschiedenen Work Prozesse über je einen eigenen Kanal mit dem Internet Communication Manager verbunden. Dieser Kanal dient dem Austausch von Ereignismeldungen zwischen dem Internet Communication Manager und den Work Prozessen und ist seinerseits wiederum auf Basis der vom Betriebssystem bereitgestellten Socket Implementierung realisiert. Außerdem besitzt der Internet Communication Manager einen auf die gleiche Weise realisierten Kanal zum Austausch von Ereignismeldungen mit dem Dispatcher.

3.5.1.1 Funktionsweise des Internet Communication Managers

Nachdem die Aufgabe des Internet Communication Managers vorgestellt wurde, soll nun noch etwas näher auf seine Funktionsweise eingegangen werden. Zum einen soll die Implementierung des Datenaustausch zwischen dem Internet Communication Manager und den Workprozessen grob umrissen werden. Zum anderen soll das Zusammenspiel zwischen Internet Communication Manager, Dispatcher und den Workprozessen anhand einiger typischer Abläufe nun noch etwas genauer vorgestellt werden.

Datenaustausch mit den Workprozessen

Wie bereits beschrieben, kommt dem Internet Communication Manager beim Datentransport über Netztwerkverbindungen die Aufgabe eines „Vermittlers“ zwischen dem Betriebssystem und den auf Basis von Workprozessen realisierten Server Side User Agents zu. Zur Weiterleitung der über die Sockets transportierten Daten an die Workprozesse wird ein „shared memory“ Bereich genutzt. Die Nutzung dieses „shared memory“ Bereichs soll nun anhand von Bild 36 etwas genauer erläutert werden. Ganz rechts im Bild ist ein Workprozess dargestellt. In seinem Innern ist der Server Side User Agent angedeutet, dessen Rolle gerade von dem Workprozess übernommen wird. Dieser Server Side User Agent kommuniziert über eine Netzwerkverbindung mit dem ganz links im Bild dargestellten Communication Partner. Die Netzwerkverbindung ist im Bild in Form des Communication Service repräsentiert, der über je einen Kanal mit dem Communication Partner und dem Internet Communication Manager verbunden ist. Diese beiden mit den Nummern 1 und 2 beschrifteten Kanäle repräsentieren dabei die beiden miteinander verbundenen Sockets über die die Kommunikation zwischen dem Server Side User Agent

24. Solche Verbindungen können prinzipiell für beliebige Zwecke genutzt werden. Beispielsweise werden sie von Anwendungsprogrammen benutzt, um Requests an andere Application Server zu übermitteln oder um auf andere Datenquellen als die zentrale Datenbank zuzugreifen.

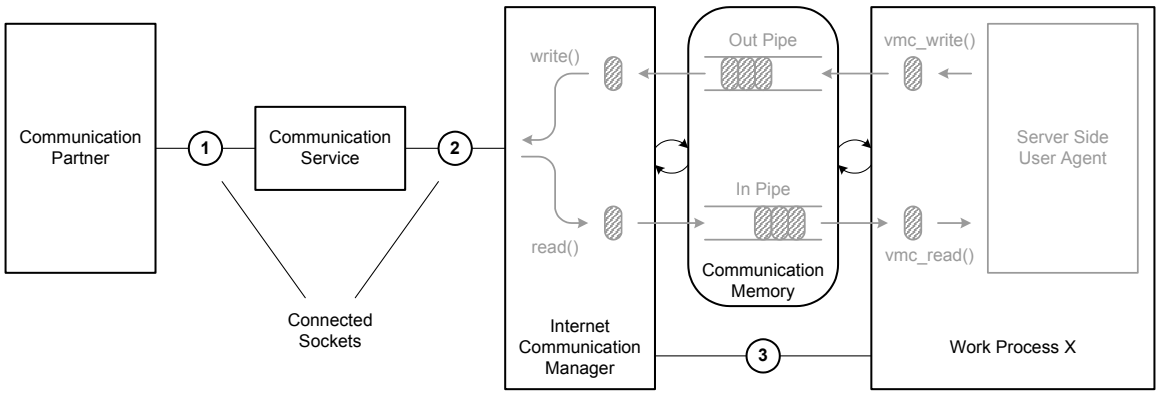


Bild 36: Internet Communication Manager - „memory pipes“

und dem Communication Partner abgewickelt wird. Der Internet Communication Manager ist seinerseits über das als „shared memory“ Bereich realisierte Communication Memory und einem Kanal mit dem betreffenden Workprozess verbunden, um die über die Netzwerkverbindung transportierten Daten mit dem betreffenden Workprozess auszutauschen. Aus Effizienzgründen geschieht der eigentliche Datenaustausch über das „shared memory“, während der Kanal, der im Bild mit der Nummer 3 beschriftet ist, nur zum Austausch von Ereignismeldungen dient. Hierzu werden für jede Netzwerkverbindung zwei Pufferbereiche im Communication Memory verwaltet, jeweils ein Puffer für ankommende Daten und ein Puffer für zu versendende Daten. Ein solches Pufferpaar ist im Communication Memory in Form einer „In Pipe“ und einer „Out Pipe“ angedeutet. Der Versand von Daten durch den Server Side User Agent erfolgt daher folgendermaßen: Zunächst schreibt der Workprozess, der gerade die Rolle des betreffenden Server Side User Agent übernimmt, die zu versendenden Daten in den der betreffenden Netzwerkverbindung zugeordneten Sendepuffer. Anschließend entnimmt der Internet Communication Manager die Daten aus dem Sendepuffer und versendet sie. Umgekehrt werden empfangene Daten vom Internet Communication Manager entgegengenommen und anschließend zur „Entnahme“ durch den Workprozess im Empfangspuffer bereitgestellt.
Der mit der Nummer 3 drei bezeichnete Ereigniskanal dient der Verteilung von „Weckrufen“. Jeder Workprozess ist über genau einen solchen Kanal mit dem Internet Communication Manager verbunden. Diese Kanäle sind im Übrigen durch zwei miteinander verbundenen Sockets (UNIX Domain) realisiert. Gäbe es diese Kanäle nicht, könnte ein Workprozess der beispielsweise darauf „wartet“, dass neue Daten über eine Netzwerkverbindung eintreffen, dies nur dadurch feststellen, dass er immer wieder überprüft, ob schon neue Daten im Empfangspuffer bereitgestellt wurden. Um solches „Polling“ zu vermeiden, tauschen die Workprozesse und der Internet Communication Manager bei Eintritt bestimmter Ereignisse, wie etwa der Ankunft neuer Daten, Weckrufe aus. Das Versenden eines Weckrufs bedeutet hierbei, dass ein Datenpaket bestimmter Größe über den betreffenden Ereigniskanal an den möglicherweise zu weckenden Prozess gesendet wird. Aufgrund der Tatsache, dass die Ereigniskanäle auf Basis von Sockets realisiert sind, kann das Eintreffen eines bestimmten Ereignisses ohne „Polling“ durch simples blockierendes Warten auf einen Weckruf (read(3)) erfolgen.

Kommunikation mit dem Dispatcher

Im letzten Abschnitt wurde nicht darauf eingegangen, dass die Rolle eines Server Side User Agents im Verlauf der Zeit von verschiedenen Workprozessen übernommen wird und dass es insbesondere auch Zeiträume gibt, während derer keiner der Workprozesse die Rolle eines bestimmten Server Side User Agents inne hat.
Damit der Internet Communication Manager seine Aufgabe erfüllen kann, muss er die Rollen-

zuordnung zwischen Server Side User Agents und Workprozessen kennen. Insbesondere braucht er dieses Wissen, um die Weckrufe an den jeweils richtigen Workprozess weiterleiten zu können. Nun ist der Dispatcher dafür zuständig, die Rollenzuordnung zwischen Server Side User Agents und Workprozessen festzulegen. Um dem Internet Communication Manager dieses Wissen zugänglich zu machen, legt der Dispatcher diese Zuordnung in einem „shared memory“ Bereich ab, auf den sowohl der Dispatcher als auch der Internet Communication Manager Zugriff haben. Dieser Speicherbereich ist im Bild 35 mit „Other Data“ bezeichnet.
Wenn der Internet Communication Manager den Bedarf für den Versand eines Weckrufs feststellt, der an einen Server Side User Agent gerichtet ist, dessen Rolle aktuell von keinem der Workprozesse gespielt wird, so informiert er den Dispatcher darüber, dass für diesen Server Side User Agent ein „Request“ vorliegt. Hierzu verwendet er den im Bild 35 dargestellten Kanal zum Dispatcher.

Behandlung von Server Sockets

Bisher wurde davon ausgegangen, dass die Verbindung zwischen dem Server Side User Agent und seinem Kommunikationspartner bereits aufgebaut ist. An dieser Stelle soll nun etwas näher auf den Aufbau von Netzwerkverbindungen eingegangen werden.
Am einfachsten ist dabei der Aufbau ausgehender Verbindungen. In diesem Fall teilt der Workprozess dem Internet Communication Manager Verbindungswunsch mit, indem er den Wunsch im Communication Memory hinterlegt und anschließend einen Weckruf an den Internet Communication Manager versendet. Der Internet Communication Manager versucht daraufhin die Verbindung herzustellen und legt bei Erfolg die zum Datentransport zwischen ICM und Workprozess(en) notwendigen Datenstrukturen für diese Verbindung im Communication Memory an. Anschließend benachrichtigt er den betreffenden Workprozess, indem er das Ergebnis des Verbindungsaufbaus im Communication Memory hinterlegt und einen Weckruf an den Workprozess versendet.
Komplizierter ist es, wenn der Verbindungsaufbauwunsch von außen an den Application Server herangetragen wird. In diesem Fall nimmt der Internet Communication Manager den auf dem entsprechenden Server Socket eintreffenden Verbindungswunsch zunächst einmal an und stellt die (TCP-)Verbindung zu dem externen Kommunikationspartner her. Nachdem die Verbindung auf Transportprotokollebene (TCP Ebene) hergestellt ist, weiß der Internet Communication Manager aber immer noch nicht, an welchen Server Side User Agent er den Verbindungswunsch weitervermitteln muss. Daher muss er in solchen Fällen zunächst die über die Verbindung eintreffenden Daten analysieren, bevor er den Verbindungswunsch an den betreffenden Server Side User Agent weiterleiten kann[25].

3.5.2 Herstellung einer VM Container tauglichen Java VM

Das Ziel dieses Abschnitts besteht darin, die Implementierung der multiplexfähigen Java VM vorzustellen, die in die bestehende Work Prozess Implementierung integriert wurde und als Basis für die Implementierung des multiplexfähigen J2EE Servers dient. Wie bereits eingangs erwähnt, wurde die CVM Implementierung der Firma Sun als Ausgangspunkt für die Implementierung dieser VM gewählt. Die Veränderungen an der CVM Implementierung betreffen im Wesentlichen die folgenden beiden Punkte:

25. Prinzipiell kann sich bei der Analyse auch herausstellen, dass der betreffende Request gar nicht an einen bestimmten Server Side User Agent gerichtet ist, sondern beispielsweise dazu dient, einen neuen Benutzer am System anzumelden. Zur Bearbeitung solcher Requests gibt es spezielle „User Contexte“.

(1) Herstellung der Multiplexfähigkeit
Einerseits muss die erforderliche Multiplexfähigkeit „eingebaut“ werden. Dies bedingt, dass der Zustand einer benutzerspezifischen Java VM nach der Bearbeitung eines Requests vom aktuellen Workprozess „abgespaltet“ und als Bestandteil des User Contextes im Context Memory des Application Servers abgelegt werden kann. Dieser im User Context enthaltene VM Zustand muss dabei so beschaffen sein, dass der Zustand der VM in einem beliebigen (anderen) Workprozess wieder „rekonstruiert“ werden kann, um weitere Requests zu bearbeiten, die diesem User Context zuzurechnen sind.

(2) Workprozesse müssen „single threaded“ bleiben
Für die Effizienz und die Skalierbarkeit ist es von entscheidender Bedeutung, dass die Workprozesse „single threaded“ sind (jedenfalls aus Sicht des Betriebssystems). Daraus ergibt sich die unmittelbare Konsequenz, dass zum Betrieb der multiplexfähigen Java VM ebenfalls nur ein Betriebssystemthread zur Verfügung steht.

In den nun folgenden Abschnitten 3.5.2.1 und 3.5.2.2 wird zunächst das grundsätzliche Vorgehen zur Umsetzung dieser beiden Punkte konkretisiert. Anschließend wird, in Abschnitt 3.5.2.3, die softwaretechnische Umsetzung sowie die Integration in das bestehende VM Container Framework grob umrissen.

3.5.2.1 Erzielung der Multiplexfähigkeit

Wie bereits erläutert, basiert die Implementierung des Workprozess Konzepts darauf, dass der User Context, also der benutzerspezifische Teil des Workprozesszustands, in einem „shared memory“ Bereich untergebracht wird, auf den alle Workprozesse zugriff haben, und dass der User Context Austausch ohne größeren Aufwand, durch Anpassung der Speicherschutzeinstellungen des betreffenden Workprozesses erreicht werden kann. Um dieses Prinzip auf die multiplexfähige Java VM zu übertragen, ist es erforderlich, sämtliche Datenbereiche, die der Realisierung der VM dienen, direkt im „shared memory“ anzulegen. Um die hierzu notwendigen Maßnahmen genauer vorzustellen, wurde der Aufbau des Adressraums eines CVM Prozesses vor und nach den Umbaumaßnahmen in Bild 37 schematisch dargestellt. Die linke Bildhälfte zeigt den Aufbau des Adressraums eines CVM Prozesses so, wie er ohne Veränderungen an der CVM Implementierung sein würde. In der rechten Bildhälfte ist die Situation nach der Durchführung der Anpassungen veranschaulicht. Der „shared memory“ Bereich, in den der benutzerspezifische Teil des VM Zustands verlagert wurde, ist dabei dunkler hinterlegt als die übrigen Teile des Adressraums.
Wie bereits in Abschnitt 2.2.2 beschrieben, benutzt die CVM für die Realisierung jedes Java Threads einen eigens dafür erzeugten Betriebssystemthread. Am oberen Ende des links dargestellten Adressraums sind daher mehrere Stack Bereiche für diese Betriebssystemthreads dargestellt. Desweiteren sind in dem Adressraum ein Heapbereich, ein Datensegment und ein Textsegment eingezeichnet. Abgesehen von einigen wenigen Daten, die im statisch allokierten Datensegment untergebracht sind, befindet sich der mit Abstand größte Teil der von der CVM verwendeten Daten im Heapbereich. Insbesondere sind dies: der Java Heap, der Methodenbereich, die Daten zur Realisierung der Java Stacks. Die Verlagerung der verschiedenen Datenbereiche in einen „shared memory“ Bereich bringt unterschiedliche Herausforderungen mit sich. Am einfachsten lies sich die Verlagerung des Heapbereichs bewerkstelligen. Hierzu musste nur eine alternative Implementierung der Speicherverwaltungsfunktionen bereitgestellt werden[26],

26. Die CVM setzt eine „POSIX memory allocation utility“ kompatible Schnittstelle voraus (<stdlib.h>: malloc(), calloc(), free(), realloc()).

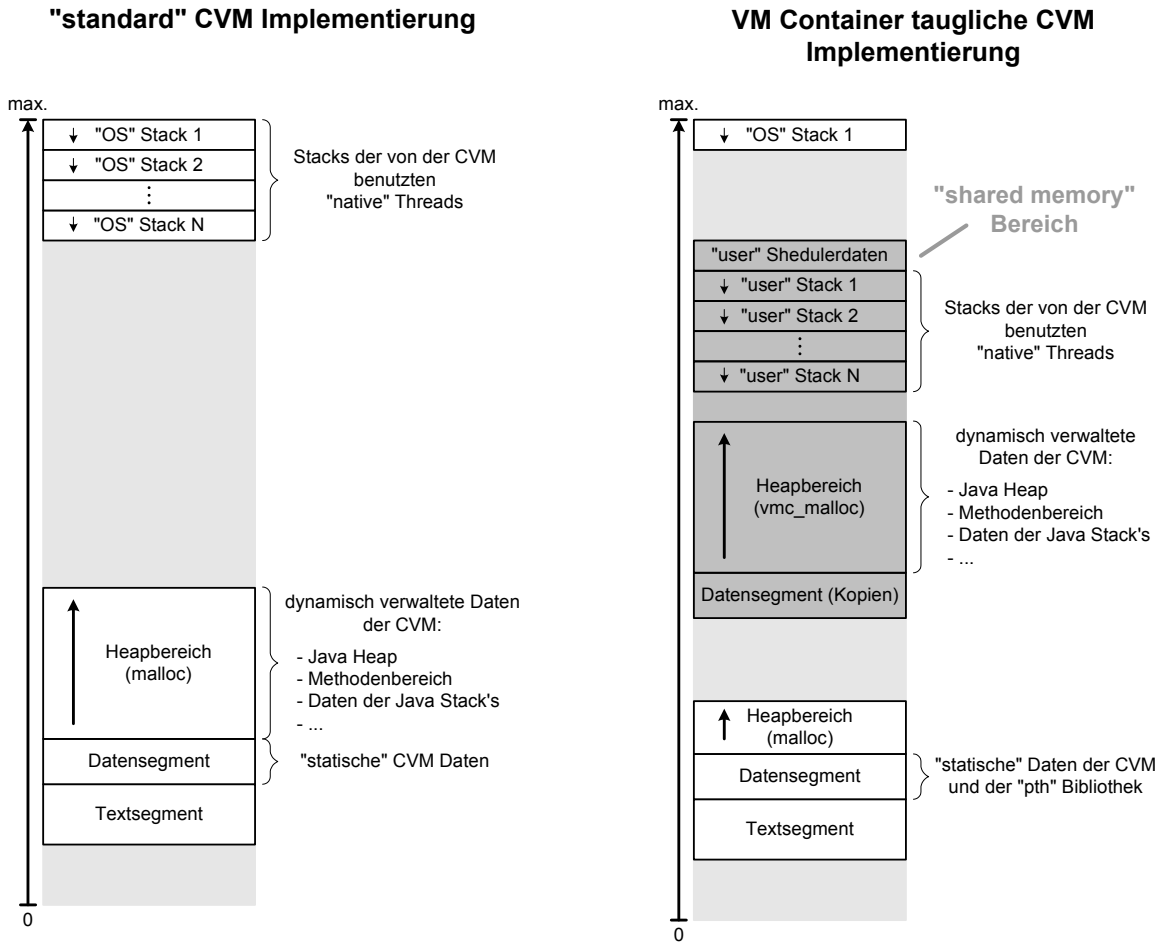


Bild 37: Aufbau des Adressraum eines CVM Prozesses (schematisch)

die den Heap direkt im „shared memory“ Bereich anlegen. Im Fall des Datensegments gestaltete sich die beabsichtigte Verlagerung in den „shared memory“ Bereich schon schwieriger, da die Erzeugung von Datensegmenten üblicherweise vom Lader/Linker des Betriebssystems übernommen wird. Da kein Weg gesehen wurde, wie die Verlagerung des Datensegments in einer portablen Weise erreicht werden kann, musste auf eine vollständige Verlagerung dieser Daten in den „shared memory“ Bereich verzichtet werden. Stattdessen wurde die Implementierung der CVM und der „native“ implementierten J2SE Class Library Methoden verändert. Diese Veränderungen zielten dabei auf zwei Dinge ab: Einerseits wurde versucht, die Verwendung von statisch allokierten Speicherbereichen durch die Verwendung von dynamisch allokiertem Speicherbereichen zu ersetzen. Andererseits wurden die verbliebenen, statisch allokierten Speicherbereiche so organisiert, dass ihr Inhalt beim Austausch des User Context „en bloc“ kopiert werden kann[27]. Auch die zum Betrieb der VM notwendigen Stack Speicherbereiche für die „native“ Threads wurden in den „shared memory“ Bereich verlagert. Die Verlagerung dieser Stack Speicherbereiche wurde dabei durch das im folgenden Abschnitt beschriebene „user space“ Scheduling stark vereinfacht. Im rechts dargestellten Adressraum ist neben den Stackbereichen, die der Realisierung der „native“ VM Threads dienen, noch der Stackbereich für den

27. Genauer gesagt, wurden die verschiedenen statischen/globalen Variablen des C Programms in einigen wenigen C-struct{}’s untergebracht, so dass ihre Inhalte beim Austausch des User Contextes mit Hilfe der memcpy() Funktion „en bloc“ kopiert werden können.

einzigen verbliebenen Betriebssystemthread dargestellt. Dieser Stackbereich enthält jedoch keine User Context spezifischen Daten.

Problem: Die in den „shared memory“ Bereich verlagerten Daten sind prozessspezifisch.

Die gerade beschriebene Verlagerung der im Adressraum des CVM Prozesses befindlichen Daten in den „shared memory“ Bereich ist, ohne weiteres Zutun, nicht geeignet, um einen prozessübergreifenden Austausch des VM Zustandes zu ermöglichen. Dieses Problem soll nun anhand von Bild 38 näher erklärt werden. Der große, grau hinterlegte Bereich soll einen CVM Betriebs-

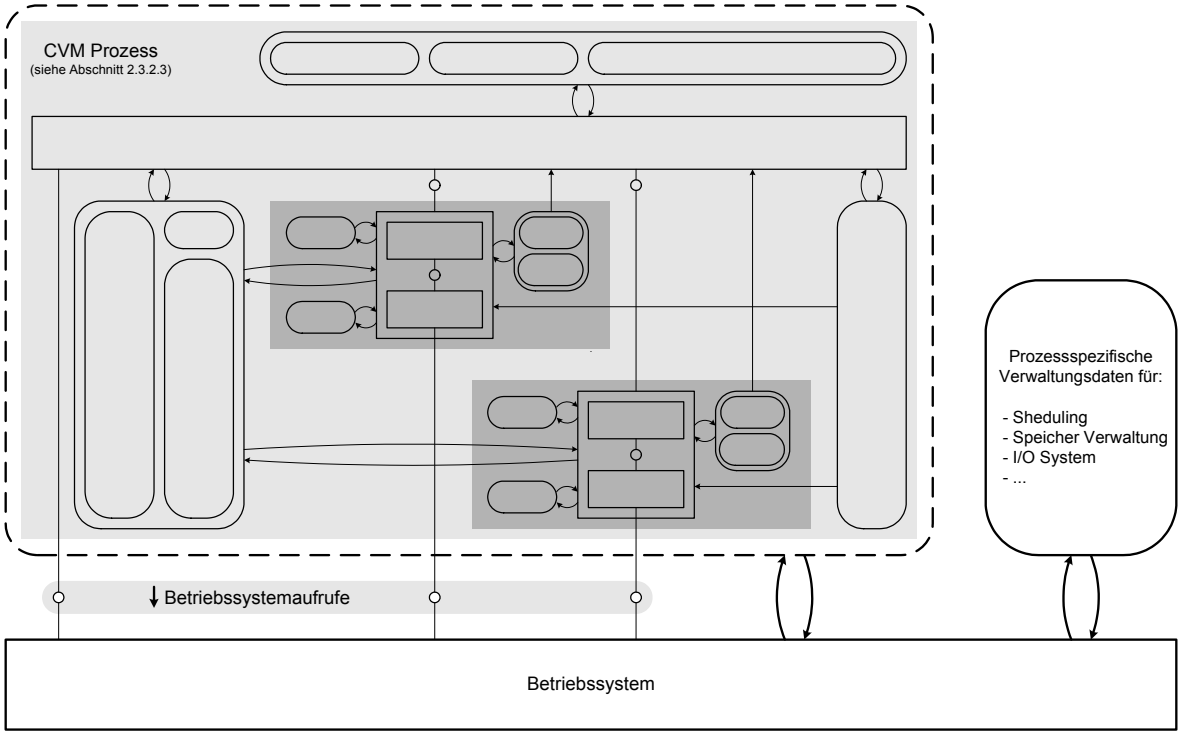


Bild 38: Bindung des CVM (Java VM) Zustands an den Betriebssystemprozess

systemprozess darstellen. Die in seinem Innern gezeigte Struktur entspricht der von Bild 14 und wurde in Abschnitt 2.2.2 bereits ausführlich vorgestellt. Da diese Struktur hier nur von sekundärem Interesse ist, wurden die Beschriftungen der Akteure und Speicher aus Gründen der Übersichtlichkeit weggelassen. Außer dem CVM Prozess ist in Bild 38 noch das Betriebssystem, in Form des unten dargestellten Akteurs, sowie ein Speicher für prozessspezifische Verwaltungsdaten dargestellt. Für die folgenden Überlegungen soll davon ausgegangen werden, dass dieser vom Betriebssystem verwaltete Speicher alle diejenigen Daten enthält, die das Betriebssystem zur Verwaltung des dargestellten CVM Prozesses benutzt. In diesem Speicher wären beispielsweise folgende Informationen zu finden:

- Prozess- / Speicherverwaltungsinformationen (Beispiele: Page Table, „parent process id“)
- Scheduling / Synchronisationsdaten (Beispiele: Thread- und Lockverwaltungsstrukturen)
- I/O Verwaltungsdaten (Beispiel: Filedeskriptortabelle)

Obwohl die Akteure des CVM Prozesses nicht direkt auf die prozessspezifischen Verwaltungsdaten zugreifen können, nehmen sie bei der Formulierung von Betriebssystemaufrufen dennoch Bezug auf diese Daten. Die im Rahmen von Betriebssystemaufrufen, zur Identifikation der ver-

schiedenen vom Betriebssystem verwalteten Objekte verwendeten Deskriptoren sind dabei prozessspezifisch und können nicht prozessübergreifend ausgetauscht werden. Um einen prozessübergreifenden Austausch des VM Zustands zu ermöglichen, muss diese Bindung zwischen den Daten, die in den „shared memory“ Bereich verlagert wurden, und dem Betriebssystemprozess aufgebrochen werden.

3.5.2.2 „single threaded“ Realisierung der Java VM

Die CVM Implementierung benutzt zur Realisierung jedes Java Threads einen eigens zu diesem Zweck erzeugten „native thread“. Die zur Erzeugung dieser „native threads“ verwendeten „POSIX Thread API“ Funktionen führen in der Standardkonfiguration der CVM dazu, dass jeder dieser „native threads“ genau einem Betriebssystemthread entspricht.
Viele Vorteile des Workprozess Konzepts resultieren aus der gezielten Beeinflussung des Schedulings. Dadurch, dass die Workprozesse im Fall des klassischen Application Servers „single threaded“ sind, kann die gewünschte Beeinflussung des Schedulings ohne explizite Unterstützung durch das zum Betrieb des Application Servers verwendete Betriebssystem erreicht werden. Die Verwendung mehrerer Betriebssystemthreads durch einen einzelnen Workprozess würde diese gezielte Beeinflussung des Schedulings ungleich aufwendiger, wenn nicht sogar unmöglich machen. Um diese Probleme zu umgehen, sollte die multiplexfähige Java VM so implementiert werden, dass zu ihrem Betrieb nur ein Betriebssystemthread erforderlich ist. Nun ist die CVM Implementierung unter der Voraussetzung entworfen worden, dass jeder Java Thread durch einen eigens dafür erzeugten „native thread“ realisiert wird. Daher muss in jedem Fall für jeden CVM „native thread“ ein eigener Prozessorkontext und ein eigener „C“-Stack verwaltet werden.
Die Grundidee, um ausgehend von der CVM Implementierung eine „single threaded“ Implementierung einer multiplexfähigen Java VM zu erstellen, bestand darin, die von der CVM Implementierung verwendeten „native threads“ durch entsprechendes „user space“ Scheduling mit Hilfe eines einzigen Betriebssystemthreads zu realisieren.

Scheduling Strategie des „user space“ Schedulers

Nun soll die multiplexfähige Java VM als Trägersystem für einen multiplexfähigen J2EE Server dienen, dessen Aufgabe in der Bearbeitung von Requests besteht. Aus diesem Grund erschien ein faires, preemptives Scheduling der „native threads“ überflüssig, denn wenn zur Bearbeitung eines Requests Beiträge von verschiedenen Java Threads geleistet werden müssen, dann müssen ohnehin alle Threads ihren Beitrag vollständig liefern, bevor die Bearbeitung des Requests beendet ist. Daher schien es zweckmäßig, die Anzahl der Kontextwechsel zu minimieren und somit auch den zur Erzielung der Fairness notwendigen Zusatzaufwand einzusparen. Bei der implementierten kooperativen Scheduling Strategie wird erst dann ein anderer Thread zur Ausführung gebracht, wenn der aktuell laufende Thread nicht länger laufbereit ist. Dass ein Thread nicht länger laufbereit ist kann dabei nur zwei Gründe haben. Entweder ist die Wartesituation dadurch begründet, dass die Erledigung eines I/O Auftrags abgewartet werden muss oder dadurch, dass eine aktuell nicht verfügbare Sperre für ein Java Objekt angefordert wurde.
In der Standardkonfiguration der CVM würden die gerade genannten Wartesituationen während der Bearbeitung von Betriebssystemaufrufen erkannt werden und das Betriebssystem würde den betreffenden Thread solange in den Wartezustand versetzen, bis der im Aufruf formulierte Auftrag fertig bearbeitet wurde.
Anhand von Bild 39 soll das Funktionsprinzip dieses „user space“ Schedulers nochmals verdeutlicht werden. Das Bild soll die Bearbeitung eines Beispielrequests veranschaulichen, an

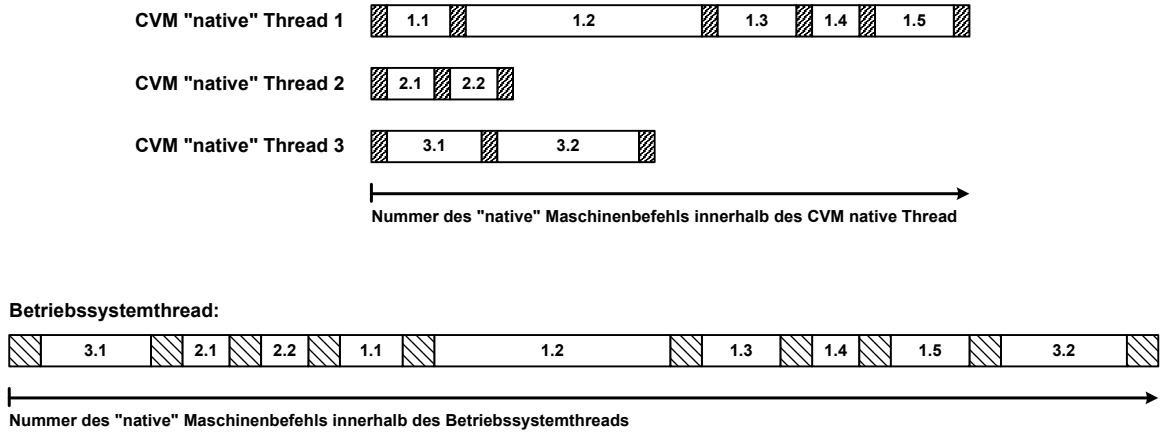


Bild 39: „user space“ Scheduling innerhalb der multiplexfähigen Java VM

dessen Bearbeitung insgesamt drei Java Threads beteiligt waren. Dem entsprechend wurde im Bild von drei CVM „native threads“ ausgegangen. Die verschiedenen Threads sind im Bild durch Balken dargestellt, die aus aneinander gereihten Rechtecken bestehen, welche abwechselnd schraffiert beziehungsweise weiß sind. In der oberen Bildhälfte sind die drei „native threads“ der CVM dargestellt, in der unteren Bildhäfte ist der zur Realisierung dieser „user space“ Threads dienende Betriebssystemthread dargestellt. Die Länge eines jeden Thread-Balkens ist dabei proportional zu der Anzahl an Maschinenbefehlen, die im Verlauf der Bearbeitung des Beispielrequests in diesem Thread ausgeführt wurden. Der einem Thread zugeordnete Balken soll in gewisser Weise ein Protokoll der ausgeführten Befehle darstellen. Die im Verlauf der Abarbeitung des Threads ausgeführten Befehle wurden dabei jeweils in zwei verschiedene Klassen eingeteilt, was sich im (Fortschritts-)Balken des Thread dadurch äußert, dass die betreffenden Abschnitte entweder schraffiert oder weiß dargestellt sind. Im Fall der CVM „native threads“ sollen die schraffierten Kästchen andeuten, dass zu den betreffenden Zeitpunkten Aufrufe solcher Funktionen durchgeführt wurden, die in der Standardkonfiguration der CVM unmittelbar durch Betriebssystemaufrufe implementiert gewesen wären. Die zwischen solchen „Pseudobetriebssystemaufrufen“ ausgeführten Befehlssequenzen, sind durch unterschiedlich lange weiße Rechtecke dargestellt. Diese in den Balken der CVM „native threads“ vorkommenden weißen Abschnitte tauchen unverändert wieder als Bestandteile des Betriebssystemthread Balkens auf. Das ist darauf zurückzuführen, dass der „user space“ Scheduler nicht preemptiv arbeitet, sondern nur dann eingreift, wenn ein „user space“ Thread nicht länger laufbereit ist. Da der Fall, dass ein „user space“ Thread nicht länger laufbereit ist, nur während der Bearbeitung von Pseudobetriebssystemaufrufen eintreten kann, wird eine zwischen zwei Pseudobetriebssystemaufrufen liegende Befehlssequenz eines „user space“ Threads niemals unterbrochen. Der Balken des Betriebssystemthread weist neben den weißen Abschnitten auch schraffierte auf. Sie stehen stellvertretend für Befehlssequenzen, die entweder dem „user space“ Scheduler zuzurechnen sind oder der Durchführung von „echten“ Betriebssystemaufrufen dienen.

Implementierungsaspekte

Zur Umstellung der CVM auf „user space“ Threads wurden sämtliche im Code der CVM vorkommenden Aufrufe von POSIX Betriebssystem API Funktionen durch Aufrufe von geeigneten Ersatzfunktionen mit gleicher Signatur ersetzt. Durch die Gleichheit der Signaturen der Original- und der Ersatzfunktionen konnte der Austausch ohne größere Änderungen am Code der CVM durchgeführt werden. Der Grund dafür, dass nicht nur die Implementierung der „POSIX Thread API“ Funktionen, sondern auch die Implementierung weiterer POSIX Betriebssys-

tem API Funktionen „ausgestauscht“ werden musste, liegt darin, dass der „user space“ Scheduler eingreifen muss, wenn im Verlauf der Bearbeitung des aktuell geschedulten „user space“ Threads (Pseudo-)Betriebssystemaufrufe durchgeführt werden sollen, die zum Blockieren des Betriebssystemthreads führen würden. In einem solchen Fall muss der „user space“ Scheduler den aktuellen „user space“ Thread in den Wartezustand versetzen und prüfen ob ein anderer „user space“ Thread laufbereit ist. Falls es einen anderen laufbereiten Thread gibt, muss der „user space“ Scheduler statt des ursprünglichen Betriebssystemaufrufs, der zur Blockade des Betriebssystemthreads geführt hätte, einen entsprechenden nicht blockierenden Aufruf durchführen und einen der anderen laufbereiten „user space“ Thread schedulen. Falls kein anderer laufbereiter „user space“ Thread existiert, wird tatsächlich ein blockierender Betriebssystemaufruf durchgeführt. Da es jedoch potenziell mehrere „user space“ Threads geben kann, die auf die Erledigung eines Betriebssystemauftrags warten, muss die mit diesem Betriebssystemaufruf verknüpfte Wartebedingung so modifiziert werden, dass der Betriebssystemthread wieder laufbereit wird, wenn irgendeiner der „user space“ Threads wieder laufbereit ist.[28]

An dieser Stelle sollen nun noch einige Bemerkungen zur Implementierung der gerade erwähnten Ersatzfunktionen gemacht werden:

- Implementierung von Synchronisationsmechanismen für „user space“ Threads
  In der Standardkonfiguration der CVM sind die von der CVM Implementierung genutzten „native threads“ unmittelbar durch Betriebssystemthreads realisiert. Ebenso basiert auch die Synchronisation dieser Threads auf der Nutzung von Betriebssystemmechanismen. Genauer gesagt werden hierzu „mutexe“ und „condition variablen“ benutzt, was sich im Code der CVM in Form von Aufrufen der entsprechenden POSIX API Funktionen äußert.
  Im Zuge der Umstellung auf „user space“ Threads mussten auch diese zur Thread Synchronisation dienenden Funktionen anders implementiert werden.
  Da alle „user space“ Threads durch einen einzigen Betriebssystemthread realisiert werden, war es bei der Implementierung dieser Ersatzfunktionen möglich, gänzlich auf die Nutzung der vom Betriebssystem bereitgestellten Thread-Synchronisationsmechanismen zu verzichten. Dieser Verzicht auf die vom Betriebssystem bereitgestellten Mechanismen, hat dabei den positiven Nebeneffekt, dass das Betriebssystem keine für die betreffende VM spezifischen Synchronisationsdaten verwalten muss und somit auch keine Abhängigkeit zwischen dem Zustand der VM und dem aktuellen Betriebssystemprozess entstehen kann.
- Implementierung des Kontextwechsels
  Der Wechsel zwischen „user space“ Threads und die damit verbundene Umschaltung des „C“ Stackbereiches erfordert den Austausch des Inhalts zahlreicher Prozessorregister. Im Fall von UNIX Systemen konnte die Implementierung dieses Kontextwechsels bemerkenswert einfach, durch Verwendung der ucontext Funktionen (siehe [ucontext 97]) realisiert werden.

VM Passivierungsstrategie / Work Prozess Umschaltung

Die Passivierung einer VM erfolgt, wenn festgestellt wird, dass sämtliche Threads der VM für „längere Zeit“ blockieren. Nun kann dies jedoch nicht ohne weiteres festgestellt werden. Speziell dann, wenn darauf gewartet wird, dass neue Daten über eine Netzwerkverbindung eintreffen, kann im Allgemeinen nicht ermittelt werden, wie lange diese Blockade anhalten wird. Nun

28. Wie in den folgenden Abschnitten noch näher erläutert wird, werden solche Wartesituationen, bei denen keiner der „user space“ Threads laufbereit ist, insbesondere auch dazu genutzt, den aktuellen User Context vom Workprozess abzutrennen.

ist es einerseits nicht zweckmäßig, eine VM sofort zu passivieren, wenn alle ihre Threads blockiert sind, da dies zu unnötigem Passivierungs-/Aktivierungsaufwand bei sehr kurzen Blockadedauern führen würde. Andererseits liegt dem Workprozess Konzept die Idee zugrunde, dass die Workprozesse „praktisch ununterbrochen arbeiten“, was erfordert, dass VM’s möglichst früh passiviert werden. Auf die Details der Passivierungsstrategie soll an dieser Stelle jedoch nicht weiter eingegangen werden, da sie für die folgenden Betrachtungen irrelevant sind.[29]

3.5.2.3 Softwaretechnische Umsetzung

Die insgesamt durchgeführten Maßnahmen, um ausgehend von der CVM Implementierung eine multiplexfähige Java VM zu erstellen, können grob in vier Bereiche gegliedert werden, die in Bild 40 durch die vier kleineren, entsprechend beschrifteten Rechtecke repräsentiert sind. Die

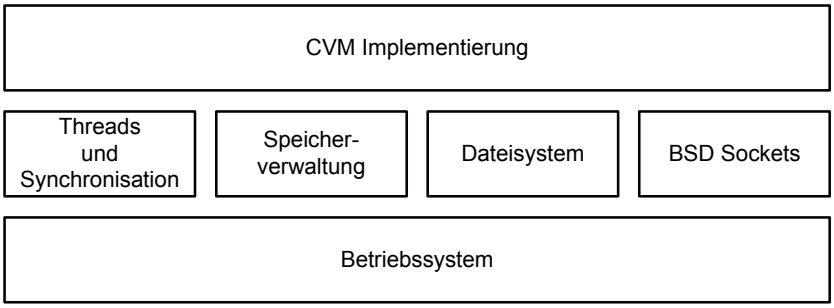


Bild 40: Herstellung der Multiplexfähigkeit durch eine zusätzliche Softwareschicht

Darstellung in Form aufeinander getürmter Rechtecke soll darauf hinweisen, dass die durchgeführten Maßnahmen darauf abzielten, die „Standard CVM Implementierung“ möglichst unverändert zu lassen. Die Herstellung der Multiplexfähigkeit sollte stattdessen durch Erstellung einer zusätzlichen Softwareschicht realisiert werden, die zwischen der CVM Implementierung und dem Betriebssystem angesiedelt ist. Wie bereits angesprochen, wurde dabei so vorgegangen, dass sämtliche im Code der CVM vorkommenden Aufrufe von POSIX Betriebssystem API Funktionen durch Aufrufe von geeigneten Ersatzfunktionen mit gleicher Signatur ersetzt wurden.
Im bisherigen Verlauf dieses Abschnitts wurde bereits ausführlich auf die Ersatzfunktionen eingegangen, die den Bereichen „Threads und Synchronisation“ und dem Bereich „Speicherverwaltung“ zuzurechnen sind. Daher wird im weiteren Verlauf dieses Abschnitts nur auf die Ersatzfunktionen zum Zugriff auf das Dateisystem und zur Nutzung von Netzwerkverbindungen (BSD-Sockets) eingegangen.

Dateisystemzugriff

Aufgrund des Multiplexbetriebs der Work Prozesse kann die vom Betriebssystem des Application Servers bereitgestellte Implementierung der im POSIX API definierten Funktionen für den Zugriff auf das Dateisystem nicht unmittelbar genutzt werden. Der Grund hierfür liegt darin, dass die zur Kommunikation mit dem Betriebssystem verwendeten Dateideskriptoren prozessspezifisch sind und daher den Workprozess-übergreifenden Austausch des VM Zustandes behindern würden. Daher wurden entsprechende Ersatzfunktionen implementiert, deren

29. Eine mögliche Passivierungsstrategie ist die Folgende: Wenn eine VM nach Eintritt in einen Blockadezustand für eine bestimmte Zeit blockiert bleibt, wird sie passiviert. Die Aktivierung der VM erfolgt erst dann wieder, wenn bei der Bearbeitung eines von ihr in Auftrag gegebenen I/O Auftrags Fortschritte erzielt werden konnten.

Funktionsweise nun unter Bezug auf Bild 41 vorgestellt wird. Die Funktionalität dieser Ersatz-

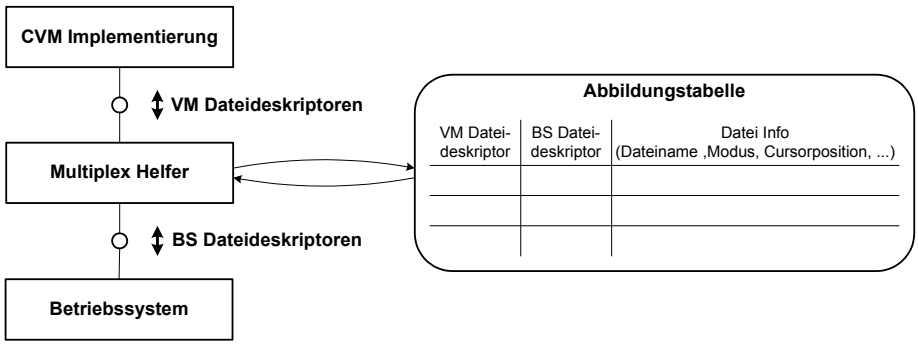


Bild 41: Abbildung von Dateideskriptoren durch den VM Container

funktionen ist im Bild in Form des „Multiplex Helfer“ Akteurs vertreten, welcher zwischen dem als „CVM Implementierung“ bezeichneten Akteur und dem Betriebssystem vermittelt. Das Funktionsprinzip des „Multiplex Helfers“ basiert auf der Idee, die zur Identifikation der aktuell geöffneten Dateien verwendeten Dateideskriptoren zu übersetzen. Wenn die CVM Implementierung beispielsweise verlangt, dass eine Datei mit einem bestimmten Namen geöffnet werden soll, gibt er diesen Auftrag unverändert an das Betriebssystem weiter. Als Ergebnis bekommt er vom Betriebssystem den zur Identifikation der geöffneten Datei vergebenen Dateideskriptor. Diesen Dateideskriptor trägt er zusammen mit dem Namen der Datei und übrigen Parametern, die die CVM Implementierung im Datei-Öffnen-Auftrag spezifiziert hatte, in eine von ihm verwaltete Abbildungstabelle ein. Des Weiteren vergibt er einen eigenen „VM Dateideskriptor“, welchen er ebenfalls in diese Tabelle einträgt. Anschließend ersetzt er den in der vom Betriebssystem gelieferten Antwort enthaltenen Dateideskriptor durch den selbst vergebenen „VM Dateideskriptor“ und liefert die so modifizierte Antwort an die CVM Implementierung zurück. Dadurch, dass der „Multiplex Helfer“ die vom Betriebssystem vergebenen Dateideskriptoren bei Datei-Öffnen-Aufträgen durch eigene „VM Dateideskriptoren“ ersetzt, verwendet die ansonsten unveränderte CVM Implementierung in nachfolgenden Aufträgen bezüglich bereits geöffneter Dateien automatisch „VM Dateideskriptoren“ zur Identifikation geöffneter Dateien. Der „Multiplex Helfer“ muss die in diesen Aufträgen enthaltenen „VM Dateideskriptoren“ wieder in die vom Betriebssystem gelieferten Dateideskriptoren „rückübersetzen“, bevor er die entsprechenden Aufträge an das Betriebssystem weiterleiten kann.
Durch die vom „Multiplex Helfer“ geleistete Abbildung sind die an die CVM Implementierung herausgegebenen „VM Dateideskriptoren“ unabhängig von den tatsächlich zur Kommunikation mit dem Betriebssystem des Application Servers verwendeten Dateideskriptoren.
Die Abbildungstabelle enthält neben den Spalten für die beiden Dateideskriptoren eine weitere als „Datei Info“ bezeichnete Spalte. Der Verwendungszweck dieser Spalte wird klar, wenn man der Frage nachgeht, was mit den Dateideskriptoren beim „Roll Out“ und „Roll In“ geschieht. Beim „Roll Out“ eines User Contextes beauftragt der „Multiplex Helfer“ das Betriebssystem mit der Schließung sämtlicher offener Dateien. Damit der „Multiplex Helfer“ nach einem erneuten „Roll In“ des betreffenden User Contextes in der Lage ist, diese Dateien erneut so zu öffnen, wie es vor dem „Roll Out“ der Fall war, trägt er die hierzu notwendige Information in die mit „Datei Info“ bezeichnete Spalte der Abbildungstabelle ein. Die Abbildungstabelle ist im Übrigen selbst Bestandteil des User Contextes und steht daher nach dem „Roll In“ automatisch wieder zur Verfügung.

Netzwerkverbindungen (BSD Sockets)

Die im Code der CVM Implementierung enthaltenen Programmteile zum Zugriff auf Netzwerkschnittstellen beinhalten entsprechende Aufrufe der im POSIX Standard definierten „BSD Socket“ Funktionen. In der Standardkonfiguration der CVM führt der Aufruf dieser Funktionen dazu, dass das Betriebssystem entsprechende Netzwerkverbindungen zwischen dem CVM-Prozess und dem jeweils gewünschten Kommunikationspartner aufbaut. Aufgrund des Multiplexbetriebs der Work Prozesse ist es im vorliegenden Fall nicht möglich, die vom Betriebssystem des Application Servers bereitgestellte Implementierung der BSD Socket-Schnittstelle direkt zu nutzen. Der Grund dafür ist, dass die Deskriptoren, die zur Identifikation der Sockets benutzt werden, ebenso wie Dateideskriptoren prozessspezifisch sind. Die Implementierung der entsprechenden Ersatzfunktionen basiert auf dem „Internet Communication Manager“ Framework. Da dieses Framework in Abschnitt 3.5.1 bereits ausführlich beschrieben wurde, soll an dieser Stelle nicht weiter darauf eingegangen werden.

3.5.3 Herstellung eines multiplexfähigen J2EE Servers auf Basis der multiplexfähigen Java VM

Da J2EE Server vollständig in Java implementiert sind, ist es von der Realisierung der im letzten Abschnitt vorgestellten multiplexfähigen Java VM hin zu einem multiplexfähigen J2EE Server kein großer Schritt mehr. Hierzu muss man letztlich nur eine vorhandene J2EE Server Implementierung nehmen und auf den im VM Container beherbergten Java VMs zur Ausführung bringen.
Im vorliegenden Fall waren dennoch einige weitere Maßnahmen notwendig, auf die nun kurz hingewiesen werden soll:

Dispatch von Requests

Bevor Clients die Dienste eines J2EE Servers in Anspruch nehmen können, müssen sie zunächst eine Netzwerkverbindung zu dem betreffenden J2EE Server aufbauen. Bei einem „normalen“ J2EE Server werden die von den Clients kommenden Verbindungsanfragen vom Betriebssystem des Rechners, auf dem der J2EE Server betrieben wird, direkt an den J2EE Server Prozess weitergeleitet. Hierzu belegt der J2EE Server Prozess beim Start des Servers die entsprechenden Ports, die von den Clients zum Verbindungsaufbau verwendet werden.
Im vorliegenden Fall ist jeder Benutzersitzung ein eigener „virtueller“ J2EE Server zugeordnet, der ausschließlich für die Bearbeitung derjenigen Requests zuständig ist, die im Verlauf dieser Benutzersitzung in Auftrag gegeben werden. Daher müssen die eintreffenden Requests vor der Zustellung an einen solchen benutzerspezifischen J2EE Server daraufhin untersucht werden, welcher Benutzersitzung sie zuzurechnen sind. Wie bereits in Abschnitt 3.5.1.1 beschrieben, fällt dies in den Aufgabenbereich des Internet Communication Managers. Dementsprechend wurde die Implementierung des Internet Communication Managers so erweitert, dass er auch solche Verbindungsanfragen behandeln kann, die an benutzerspezifische J2EE Server gerichtet sind. Ferner mussten die mit der Nutzung von Server-Sockets in Zusammenhang stehenden Teile der Implementierung der verwendeten Web- und EJB-Container angepasst werden, um die gewünschte Integration in das VM Container Framework zu erreichen.

Betriebsmittelverwaltung durch Web- und EJB-Container

Eine Implementierung eines J2EE Servers wird normalerweise im Hinblick darauf entworfen, dass ein J2EE Server durch viele Benutzer genutzt wird. Die Vorgehensweise, jedem Benutzer

einen eigenen J2EE-Server zur Verfügung zu stellen, ist daher nicht ganz unproblematisch. Im Rahmen der durchgeführten Tests ergaben sich daraus jedoch keine gravierenden Probleme. Dies ist nicht zuletzt darauf zurückzuführen, dass viele Probleme durch geeignete Konfigurationseinstellungen umgangen werden konnten.

Hintergrund-Threads

Obwohl die Aufgabe eines J2EE Servers primär in der Bearbeitung der an ihn gerichteten Requests besteht, bedeutet dies nicht, dass sämtliche zur Realisierung des Servers dienende Java Threads blockiert sind, wenn aktuell keine zu verarbeitende Requests vorliegen. Stattdessen ist es für J2EE Server typisch, dass sie in regelmäßigen Abständen bestimmte Aktionen durchführen, die nicht mit der Verarbeitung eines Requests in Zusammenhang stehen. Beispielsweise überprüfen J2EE Server im Abstand weniger Sekunden, ob Änderungen an den Konfigurationsdateien des Servers durchgeführt wurden und lesen die Konfigurationsdateien gegebenenfalls neu ein. Solche dauernd laufenden Threads haben sich im vorliegenden Fall als problematisch erwiesen, weil sie einen erhöhten Bedarf an „User Context“ Wechseln nach sich ziehen.

4 Integration von virtuellen Maschinen in Systeme

Diese Arbeit beschäftigt sich mit der Integration virtueller Maschinen. Die im letzten Kapitel beschriebene Integration einer Java VM in den SAP Application Server ist dabei nur ein Beispiel für eine solche Integration. In diesem Kapitel soll das Problem der Integration virtueller Maschinen in Systeme nun allgemeingültiger behandelt werden und der Bezug zu anderen in der Literatur beschriebenen Forschungsaktivitäten hergestellt werden. Aufgrund der großen Vielfalt an Integrationsszenarien und virtuellen Maschinen ist es unmöglich, das Spektrum der dabei auftretenden Problemstellungen vollständig darzustellen. Dennoch gibt es verschiedene typische Problemfelder, die für eine Vielzahl an Integrationsszenarien relevant sind.

Die nun folgende Vorstellung dieser Problemfelder gliedert sich in zwei Teile. Zunächst werden in Abschnitt 4.1 weitere Integrationsbeispiele vorgestellt. In dem sich daran anschließenden Abschnitt 4.2 wird dann näher auf die angesprochenen Problemfelder eingegangen. Die Beispiele in Abschnitt 4.1 dienen in erster Linie als Argumentationsgrundlage für die Vorstellung der Problemfelder. Um ein möglichst breites Spektrum an typischen Problemstellungen diskutieren zu können, handelt es sich bei den vorgestellten Beispielen um Systeme mit unterschiedlichen Aufgabenstellungen.

Da die bei der Integration auftretenden Problemstellungen in Abschnitt 4.2 ausführlich diskutiert werden, wird bei der Vorstellung der einzelnen Beispiele nicht oder nur kurz auf die mit dem betreffenden Beispiel einhergehenden Probleme hingewiesen. Die Vorstellung der Beispiele konzentriert sich daher auf die Darstellung des durch die Integration der virtuellen Maschinen erreichten Mehrwerts.

4.1 Beispiele für die Integration von VM's in Systeme

4.1.1 Web-Browser

Die erste Generation von Web-Browsern ermöglichte die Darstellung von Webseiten, die aus unterschiedlich formatierten Texten und aus Bildern im GIF Format bestanden. Die entsprechenden Webseiten hatten daher ein „statisches" Erscheinungsbild, und die Interaktion mit dem Browser war im Wesentlichen auf das Verfolgen von Hyperlinks beschränkt. Sehr bald nach dem Erscheinen der ersten Web-Browser kam der Wunsch nach leistungsfähigeren Browsern auf, die in der Lage waren, animierte Seiteninhalte (animierte Graphiken, Newsticker, ...) darzustellen. Im Übrigen wurden Web-Browser auch schon sehr bald zur Realisierung von Clients für Client-Server Systeme eingesetzt. Um diesen Ansprüchen gerecht zu werden, wurden die Fähigkeiten der Web-Browser in der Folge stetig erweitert. Ein Beispiel für ein erfolgreiches Konzept, das der Erweiterung der Fähigkeiten von Browsern dient, ist das Konzept der „Applets"[30]. Ein Applet ist ein Java Programm, das über eine graphische Benutzeroberfläche verfügt und vom Browser automatisch ausgeführt wird, wenn die betreffende Webseite dargestellt werden soll. Den Programmcode des Applet lädt der Browser ebenso wie alle übrigen zur Darstel-

30. Der erste Browser, der Applets unterstützte, war der 1995 von der Firma Netscape entwickelte „Netscape 2.0" Web Browser.

lung der betreffenden Seite notwendigen Daten von dem entsprechenden Web-Server herunter. Die von einem solchen Applet erzeugte graphische Darstellung wird vom Browser, ähnlich wie ein Bild, als Bestandteil der betreffenden Webseite dargestellt. Bild 42 zeigt eine Webseite, die

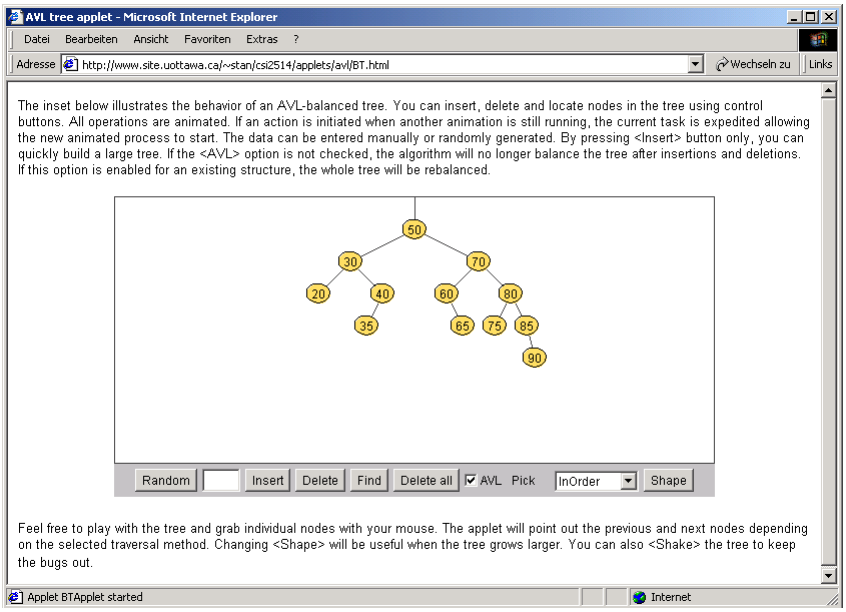


Bild 42: Webseite mit Applet

ein Applet enthält. Das im Bild dargestellte Beispielapplet dient dazu binäre Suchbäume zu visualisieren. Die Benutzeroberfläche des Applet nimmt, wie für Applets kennzeichnend, einen rechteckigen Bereich fester Größe innerhalb der Webseite ein und besteht aus herkömmlichen Bedienelementen wie Buttons, Textfeldern, Check-Boxen, ... Wenn der Benutzer der Webseite über die Bedienelemente das Einfügen eines neuen Schlüssels in den Baum anstößt, berechnet das Applet eine Folge von Bildern, die dem Benutzer in Form einer kurzen Animation dargeboten wird und die der Veranschaulichung der einzelnen Schritte des Einfügeprozesses dient.
Ein Browser, der die Nutzung von Applets unterstützt, muss über die Möglichkeit verfügen, Java Code auszuführen. Die Java VM, die zur Ausführung des Applet Codes benutzt wird, ist dabei üblicherweise nicht als separater Betriebssystemprozess realisiert, sondern ist normalerweise direkt in den Browser integriert. Die Form der Integration der Java VM soll nun anhand von Bild 43 kurz vorgestellt werden. In der linken Bildhälfte ist ein Web-Browser dargestellt. Zur Kommunikation mit den rechts dargestellten Web Servern benutzt er das HTTP-Protokoll. Die innere Struktur des Web-Browsers besteht aus drei Akteuren: dem „User I/O Agent“, dem „Download and Control Agent“ und einer Java VM. Die links dargestellte Java VM dient zur Ausführung der Applet Programme. Der „User I/O Agent“ ist für die graphische Darstellung von Webseiten am Bildschirm und für die Entgegennahme von Benutzereingaben zuständig. Der „Download and Control Agent“ ist für die Kommunikation mit den Web-Servern zuständig. Er lädt die zur Darstellung der Webseiten notwendigen Daten von den betreffenden Web-Servern herunter und stellt sie den anderen beiden Akteuren im Speicher „Downloaded Data“ zur Verfügung.
Das innerhalb der Java VM des Browsers nicht nur ein Applet Akteur, sondern mehrere Applet-Akteure angedeutet sind, hat dabei zwei Gründe: Einerseits kann eine Webseite mehrere Applets beinhalten, was dazu führt, dass alle zu der Seite gehörenden Applet Programme gestartet werden, wenn die betreffende Webseite dargestellt werden soll. Andererseits wird ein Web-Browser typischerweise dazu benutzt, nacheinander verschiedene Webseiten anzuzeigen, wobei jede Webseite Applets enthalten kann. Unabhängig davon, ob mehrere Applets gleichzeitig oder

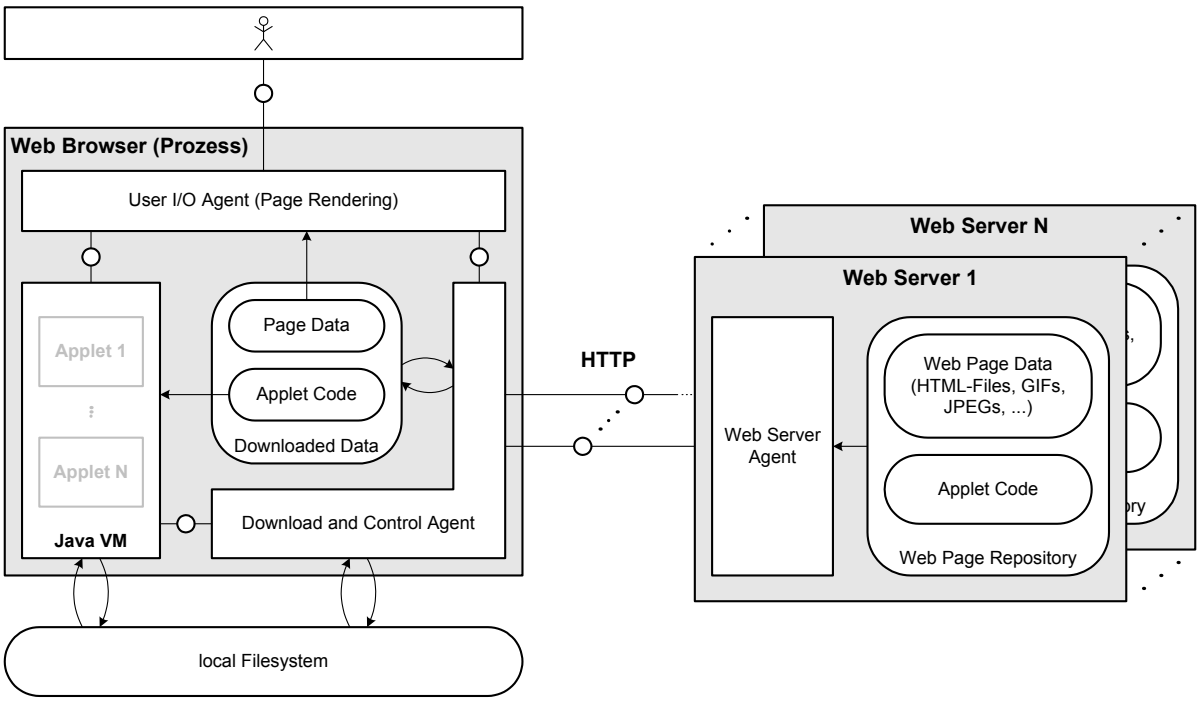


Bild 43: Integration einer Java VM in einen Web-Browser

stets nur ein Applet ausgeführt wird, benutzt ein Web-Browser normalerweise nur eine einzige Java VM zur Ausführung sämtlicher Applets, die im Verlauf der Benutzung des Browsers ausgeführt werden.

4.1.2 Integration in relationale Datenbanksysteme - „stored procedures“

Die Sprache SQL hat sich zum „de facto“ Standard zur Formulierung von Anfragen an relationale Datenbanksysteme entwickelt. Die Stärke von SQL liegt im Wesentlichen in der Möglichkeit zur Formulierung komplexer Datenbankabfragen. Solche Abfragen werden in Form von Ausdrücken formuliert, die vom Datenbanksystem ausgewertet werden und als Ergebnis eine Tupelmenge liefern. Während sich in SQL sehr komplexe Abfragen formulieren lassen, bietet SQL nur eingeschränkte Möglichkeiten zur Formulierung von Operationen zur Veränderungen des Datenbankinhalts. Insbesondere verfügt SQL nicht über imperative Sprachelemente zur Formulierung von Befehlssequenzen, Fallunterscheidungen und Schleifen. Durch diese Einschränkung können Operationen zur Veränderung des Datenbankinhalts häufig nur mit Hilfe mehrerer Datenbankabfragen formuliert werden. Die damit verbundenen Nachteile sollen nun am Beispiel des in Bild 44 dargestellten Informationssystems einer Bank verdeutlicht werden. Als Beispielablauf soll der Ablauf zur Überweisung von Geldbeträgen zwischen verschiedenen Girokonten dienen. Ganz links im Bild sind die Kunden der Bank dargestellt. Sie können die von der Bank bereitgestellten Überweisungsterminals nutzen, um Überweisungen in Auftrag zu geben. Jedes Überweisungsterminals ist mit einem Buchungsserver verbunden. Ein solcher Buchungsserver ist für die Bearbeitung sämtlicher Überweisungsaufträge zuständig, die an den angeschlossenen Terminals in Auftrag gegeben werden. Die eigentlichen Informationen über die Konten sind in einem zentralen (relationalen) Datenbanksystem abgelegt. Damit die Buchungsserver die Überweisungsaufträge bearbeiten können, stehen sie mit dem zentralen Datenbanksystem der Bank in Verbindung. Die zentrale Datenbank enthält unter anderem eine Tabelle „Girokonto“, die in der unteren Hälfte von Bild 44 veranschaulicht ist und je einen Eintrag für

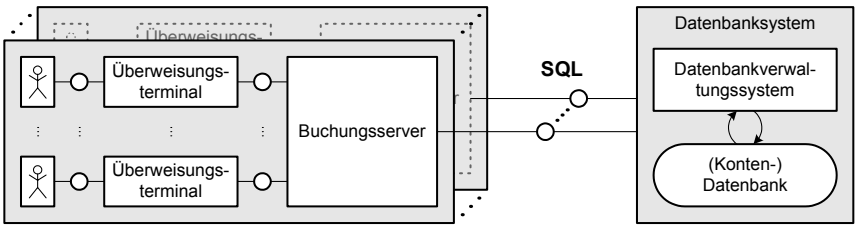


Tabelle "Girokonto" :

Kontonummer	Kontostand	Dispo-Kreditrahmen	...
111111111	*1.154,50*	*1.500,00*	
222222222	*247.546,00*	*10.000,00*	
⋮	⋮	⋮	⋮

Bild 44: „On-Line Transaction Processing“ - Bankbeispiel

jedes von der Bank verwaltete Girokonto enthält. Die Einträge können über die Kontonummer identifiziert werden und geben unter anderem Auskunft über den aktuellen Kontostand sowie über die Höhe des eingeräumten (Dispo-) Kreditrahmens.
Gibt ein Kunde an einem der Überweisungsterminals eine Überweisung in Auftrag, so muss vor der Durchführung des Überweisungsauftrags zunächst geprüft werden, ob das Konto dieses Kunden nach der Überweisung ein Soll aufweisen würde, welches den eingeräumten Kreditrahmen überschreitet. In diesem Fall muss die Überweisung zurückgewiesen werden.
Aufgrund der beschränkten Sprachmächtigkeit von SQL muss der Buchungsserver zur Bearbeitung eines Überweisungsauftrags mehrere Anfragen an das Datenbanksystem stellen. Eine Anfrage um zu überprüfen, ob der Überweisungsauftrag durchgeführt werden kann und gegebenenfalls zwei weitere, um die Kontostände der beteiligten Konten anzupassen. Diese Anfragen müssen überdies zu einer Transaktion zusammengefasst werden. Da der Buchungsserver die beiden letzten Anfragen frühestens nach Erhalt des Ergebnisses der ersten Abfrage abschicken kann, nimmt eine solche Überweisungstransaktion mindestens doppelt soviel Zeit in Anspruch, wie zur Übertragung einer Nachricht zwischen Buchungsserver und Datenbanksystem erforderlich ist.
Die Belastung des zentralen Datenbanksystems könnte im gerade beschriebenen Beispiel wesentlich reduziert werden, wenn die zur Bearbeitung einer Überweisung notwendigen Anfragen zu einer einzigen Anfrage zusammengefasst werden könnten. Dies würde einerseits den Kommunikationsbedarf zwischen den Buchungsservern und dem zentralen Datenbanksystem erheblich reduzieren, andererseits könnten die zur Bearbeitung der Überweisungstransaktion erforderlichen Betriebsmittel wesentlich schneller wieder freigegeben werden, da zwischen Beginn und Ende der Überweisungstransaktion keine durch den Nachrichtenaustausch bedingten „Wartepausen“ entstehen würden.

Da das vorgestellte Problem nicht neu ist, unterstützt die Mehrzahl aktueller Datenbanksysteme das Konzept der „stored procedures“. Solche „stored procedures“ sind Programmstücke, die beim Datenbanksystem hinterlegt werden und auf Nachfrage vom Datenbanksystem ausgeführt werden. Ähnlich wie die aus prozeduralen Sprachen bekannten Prozeduren können „stored procedures“ auch Parameter besitzen. Würde das in Bild 44 dargestellte Datenbanksystem „stored procedures“ unterstützen, so könnte der Ablauf zur Bearbeitung eines Überweisungsauftrags dort in Form einer „stored procedure“ hinterlegt werden. Diese „stored procedure“ würde als Parameter die Kontonummern der beteiligten Konten sowie den zu überweisenden Betrag haben. Zur Bearbeitung eines Überweisungsauftrags durch einen Buchungsserver wäre dann tat-

sächlich nur noch eine einzige Anfrage an das zentrale Datenbanksystem notwendig, in der die Ausführung der betreffenden „stored procedure“ mit den passenden Parametern verlangt wird. In der Vergangenheit mussten „stored procedures“ in speziell zu diesem Zweck entwickelten Datenbanksystem-spezifischen Sprachen formuliert werden. Inzwischen bieten führende Hersteller (darunter IBM und Oracle) Datenbanksysteme an, bei denen „stored procedures“ in Java formuliert werden können. Zur Ausführung dieser in Java formulierten „stored procedures“ sind in solche Datenbanksysteme Java VM’s integriert.

Anhand von Bild 45 soll nun näher auf die Form der Integration von Java VM’s in solche Da-

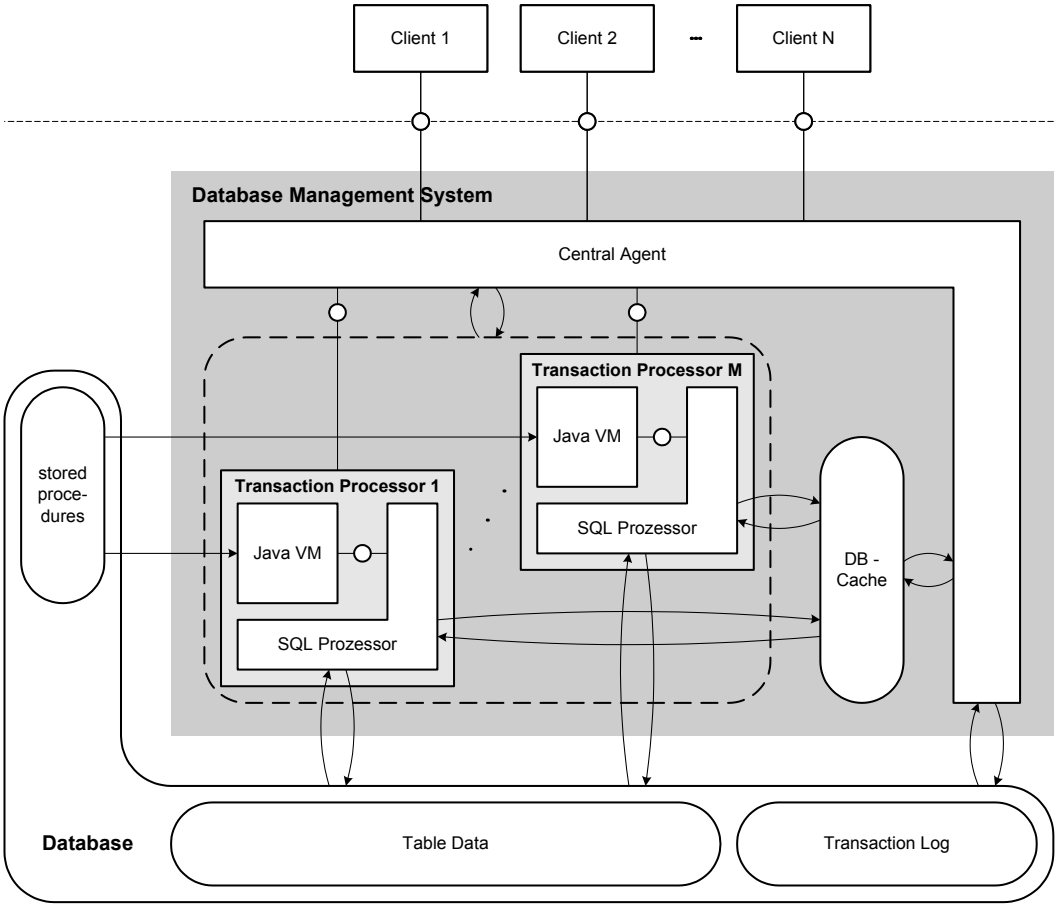


Bild 45: Integration von Java VM’s in relationale Datenbanksysteme

tenbanksysteme eingegangen werden. Das eigentliche Datenbanksystem ist im unteren Teil des Bildes dargestellt. Es besteht aus dem Datenbankverwaltungssystem, welches im Bild grau hinterlegt ist und einem „L“-förmig dargestellten Speicher, der die Datenbank darstellt. Oberhalb des Datenbanksystems sind die Clients dargestellt, die mit Hilfe des Datenbankverwaltungssystems auf die Datenbank zugreifen. Die Datenbank umfasst neben den als „Table Data“ bezeichneten „Nutzdaten“ auch die beim Datenbanksystem hinterlegten „stored procedures“ sowie das Transaktionsprotokoll. Das Datenbankverwaltungssystem besteht seinerseits aus einem „Central Agent“, einem als „DB-Cache“ bezeichneten Puffer für Datenbankinhalte sowie einer variablen Anzahl von „Transaction Processor“ Akteuren. Dem in Bild 45 dargestellten Modell liegt dabei die Vorstellung zugrunde, dass zu Beginn jeder Transaktion ein neuer „Transaction Processor“ erzeugt wird. Dieser „Transaction Processor“ ist für die Bearbeitung sämtlicher zu dieser Transaktion zusammengefassten Anfragen zuständig und wird nach Abschluss der Transaktion wieder zerstört. Ein solcher „Transaction Processor“ besteht seinerseits aus einem SQL Prozessor für die Bearbeitung von „normalen“ SQL Anfragen und einer Java VM zur Aus-

führung von „stored procedures“. Von diesen beiden Akteuren kann nur der SQL Prozessor direkt auf den Datenbankinhalt und den DB-Cache zugreifen. Er ist insbesondere auch für die Auswertung der im Rahmen der Ausführung von „stored procedures“ erforderlichen Datenbankzugriffe zuständig. Aus Sicht des Programmierers einer „stored procedure“ gibt es üblicherweise keine Unterschiede zwischen der Formulierung von Zugriffen auf das „lokale“ Datenbanksystem und Zugriffen auf entfernte Datenbanksysteme. Sie werden ebenso wie Abfragen an entfernte Datenbanksysteme in Form von (Standard) „SQL“-Abfragen formuliert, die an das Datenbanksystem „gesendet“ werden. Die im Rahmen der Ausführung von „stored procedures“ stattfindenden Zugriffe auf den vom Datenbanksystem verwalteten Datenbestand unterliegen dabei ebenso dem Transaktionsschutz wie alle übrigen Datenbankzugriffe, die im Verlauf einer Transaktion durchgeführt werden. Bei der im Zusammenhang mit dem Bankbeispiel vorgestellten „stored procedure“ war die Transaktionsdauer mit der Ausführungsdauer der „stored procedure“ identisch. Üblicherweise ist die Transaktionsdauer jedoch von der Ausführungsdauer einer „stored procedure“ entkoppelt, so dass im Verlauf einer Transaktion einerseits mehrere „stored procedures“ zur Ausführung kommen können. Andererseits können im Verlauf einer Transaktion sowohl normale SQL Anfragen als auch solche Anfragen gestellt werden, die dem Aufruf von „stored procedures“ dienen.

4.1.3 Orthogonal Persistent Java - PEVM

Klassischen Anwendungssystemen, wie zum Beispiel Textverarbeitungssystemen oder CAD-Zeichenprogrammen, liegt eine klare Trennung zwischen dem „eigentlichen Anwendungsprogramm“ und der mit Hilfe des Programms bearbeiteten „Dokumente“ zugrunde. Der Nutzungsprozess eines solchen „Anwendungsprogramms“ kann üblicherweise in folgende Phasen unterteilt werden: „Anwendung starten“, „Dokument öffnen“, „Nutzen der Anwendung zur Bearbeitung des Dokuments“, „Abspeichern des Dokuments“, „Schließen der Anwendung“. Mit diesem Nutzungsprozess geht normalerweise ein Prozess der Umcodierung des zu bearbeitenden Dokuments einher: Beim „Öffnen des Dokuments“ werden die in der Dokument-Datei enthaltenen Daten eingelesen und in einer für die Bearbeitung des Dokuments geeigneten Arbeitscodierung im Hauptspeicher bereitgehalten. Entsprechend werden die im Hauptspeicher vorhandenen, in der Arbeitscodierung vorliegenden Daten beim „Abspeichern des Dokuments“ wieder in eine Konservierungscodierung überführt und in eine Datei geschrieben. Die Entwicklung der Arbeitscodierung, der Konservierungscodierung sowie der Umcodierungsalgorithmen ist normalerweise die Aufgabe des Anwendungsentwicklers. Die Entwicklung geeigneter Codierungen und Umcodierungsalgorithmen stellt dabei oft einen nicht zu vernachlässigenden Entwicklungsaufwand dar. Der Entwicklungsaufwand ist umso größer, wenn die Daten des zu bearbeitenden Dokuments derart umfangreich sind, dass nicht alle Daten gleichzeitig im Hauptspeicher bereitgestellt werden können, sondern immer nur ein bestimmter Anteil dieser Daten aktuell im Hauptspeicher gehalten werden kann.
Die in diesem Abschnitt vorgestellte PEVM ist eine Implementierung einer „**O**rthogonal **P**ersistent **J**ava Plattform (OPJ)“. Mit der Entwicklung von OPJ-Plattformen wird das Ziel verfolgt, die Entwicklung von anwendungsspezifischen Konservierungscodierungen überflüssig zu machen. Hierzu verfügen solche VMs über einen eingebauten Checkpoint-Mechanismus, der es erlaubt, den Zustand der aktuell in Ausführung befindlichen Java-Anwendung „einzufrieren“ und in eine Datei zu schreiben. Der Inhalt der im Rahmen einer „Checkpoint“ Operation erzeugten Datei wird dabei als „suspended computation“ bezeichnet und kann verwendet werden, um die Anwendung später, möglicherweise auf einem anderen Rechner, wieder zu aktivieren („resume“). Da eine solche „suspended computation“ auch denjenigen Teil des VM Zustands umfasst, der das aktuell in Bearbeitung befindliche Dokument repräsentiert, stellt eine „suspended com-

putation“ gleichzeitig auch eine Konservierungscodierung des zum Zeitpunkt der „Checkpoint“ Operation in Bearbeitung befindlichen Dokuments dar.
Die bei der Implementierung einer OPJ-VM zu lösenden Probleme weisen eine gewisse Ähnlichkeit mit denjenigen Problemen auf, die im Zusammenhang mit der Realisierung der in Abschnitt 3.5.2 vorgestellten „multiplexfähigen Java VM“ diskutiert wurden, denn in beiden Fällen geht es darum eine VM zu konstruieren, die es erlaubt, den aktuellen VM Zustand so abzuspeichern, dass er später wieder rekonstruiert werden kann, um die unterbrochene Berechnung fortzusetzen. Da sich die zur Anwendung kommenden Lösungsansätze aufgrund der unterschiedlichen Rahmenbedingen dennoch stark unterscheiden, soll anhand von Bild 46 noch

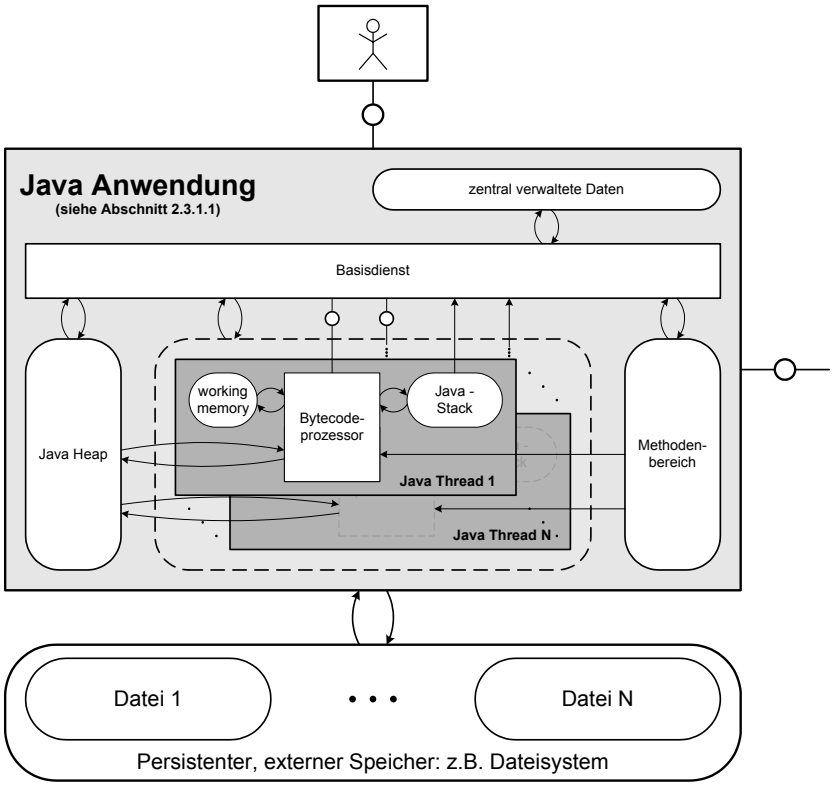


Bild 46: „Orthogonal Persistence“ for Java

einmal neu auf die Probleme eingegangen werden. In der Mitte des Bildes ist ein Akteur dargestellt, der eine in Java programmierte Anwendung repräsentieren soll. In seinem Innern ist die in Abschnitt 2.2.1.1 vorgestellte Struktur einer Java VM angedeutet. Die Java VM, die der Realisierung des Anwendungsakteurs dient, besitzt einen Kanal zum Benutzer, hat Zugriff auf das am unteren Bildrand in Form eines Speichers dargestellte Dateisystem und unterhält möglicherweise Netzwerkverbindungen zu anderen Akteuren, sofern dies vom Anwendungsprogramm verlangt wird.
Das OPJ Konzept sieht vor, dass der gesamte Zustand einer Java VM im Rahmen einer Checkpoint Operation in eine Datei geschrieben wird. Der VM Zustand setzt sich dabei aus den Inhalten verschiedener Speicher zusammen: Java Heap, Methodenbereich, Stackbereiche der verschiedenen Java Threads und den zentral verwalteten Daten. Nun ist die bloße Sicherung des Inhalts der gerade aufgezählten Speicherbereiche nicht ausreichend, um den Anwendungsakteur später wieder auf einem anderen Rechner reaktivieren zu können. Selbst wenn man, dem OPJ-Gedanken folgend, davon ausgeht, dass der Anwendungsakteur das lokale Dateisystem nicht nutzt, sondern alle von ihm verwalteten Daten in Form entsprechender Objekte im Java Heap ablegt, so besteht bei der Reaktivierung einer „suspended computation“ dennoch das Pro-

blem, die Kanäle zum Benutzer und die Netzwerkverbindungen zu anderen Akteuren wieder herzustellen. Da sich dieses Problem nicht allgemeingültig auf VM Ebene lösen lässt, sieht die OPJ-Spezifikation einen Mechanismus vor, um dem Anwendungsprogrammierer die Aufgabe des Abbaus und der Wiederherstellung solcher Kanäle zu übertragen.

Bei herkömmlichen Java VM's sind die Speicher für den Java Heap, den Methodenbereich, die Stacks der Java Threads sowie der Speicher für zentral verwaltete Daten auf Basis entsprechender Hauptspeicherbereiche realisiert. Würde man eine OPJ-VM in gleicher Weise implementieren, so wäre die Nutzung dieser OPJ-VM auf Anwendungsfälle beschränkt, bei denen die gesamten Anwendungsdaten vollständig im Hautspeicher untergebracht werden können. Um den mit den OPJ Konzept verbundenen Vorteil auch dann nutzen zu können, wenn die mit Hilfe einer Anwendung bearbeiteten „Dokumente" so umfangreich sind, dass sie nicht vollständig in den Hauptspeicher passen, verfügt die PEVM über einen speziellen Auslagerungsmechanismus. Dieser Mechanismus basiert auf der Idee, bei einem Neustart einer „suspended computation" nicht gleich alle in dieser „suspended computation" enthaltenen Heap Objekte und Klassendefinitionen in den Hauptspeicher zu laden, sondern diesen Ladevorgang so lange wie möglich zu verzögern und die entsprechenden Daten erst dann zu laden, wenn bei der Programmausführung erstmalig darauf zugegriffen wird. Dieser Mechanismus hat gewisse Ähnlichkeit mit dem aus der Betriebssystemwelt bekannten Paging Mechanismus. Der Hauptunterschied zum Paging besteht darin, dass einmal eingelagerte Daten, die nach ihrer Einlagerung verändert wurden, zwangsläufig bis zum nächsten Checkpoint im Hauptspeicher verbleiben müssen. Ferner erfolgt die Ein-/Auslagerung von Daten nicht in der Granularität ganzer Hauptspeicherseiten. Vielmehr ist die PEVM so konstruiert, dass jedes Objekt des Java Heap einzeln auslagert werden kann. Ein weiterer Unterschied zum Paging ist, dass der Gesamtumfang des von einer PEVM verwalteten Java Heaps um mehrere Größenordnungen höher sein darf als derjenige Teil, der im Hauptspeicher untergebracht werden kann. Daher kann der für Heap-Daten reservierte Teil des Hauptspeichers als Cache für Heap Inhalte betrachtet werden.

Unter Bezug zu Bild 47 soll die Funktionsweise der PEVM nun noch etwas genauer umrissen werden. Der grau hinterlegte Bereich in der oberen Bildhälfte zeigt ein sehr einfaches Modell der PEVM.[31] Darunter ist ein als Persistent Object Store bezeichneter Speicher dargestellt, der die Daten einer „suspended computation" enthält. Im Fall der PEVM ist dieser Speicher durch eine Datei realisiert.
Im dargestellten Modell der PEVM wird erneut die im vorangegangenen Bild verwendete, aus Abschnitt 2.2.1.1 bekannte Grundstruktur einer Java VM deutlich. Die Ähnlichkeit zum vorherigen Bild erkennt man leicht an den in beiden Bildern vorkommenden, dunkelgrau hinterlegten Java Thread Akteuren, die rechts und links von Speichern und oben vom Basisdienstakteur umgeben sind. Bei dem in Bild 46 dargestellten Modell enthält der rechts von den Java Thread Akteuren dargestellte Speicher sämtliche Informationen über alle von der VM geladene Klassen, und der links davon dargestellte Speicher enthält den gesamten Java Heap. Im Fall der PEVM befindet sich jeweils nur ein Teil dieser Daten tatsächlich im Hauptspeicher. In Bild 47 repräsentieren die rechts und links von den Java Thread Akteuren dargestellten Speicher die Ablageorte für die aktuell im Hauptspeicher befindlichen Teile des Java Heap und des Methodenbereichs. Dementsprechend sind die beiden Speicher dort als Heap Cache und Method Cache bezeichnet. Neu hinzugekommen ist der als „Store Driver" bezeichnete Akteur, wel-

31. Die PEVM ist eine modifizierte Version der „Sun Microsystems Laboratories Virtual Machine for Research (Research VM)". Die Research VM selbst ist eine herkömmliche Java VM und beinhaltet neben einem Interpreter auch einen optimierenden Compiler. Da das Vorhandensein des optimierenden Compilers kaum Einfluss auf die durchzuführenden Änderungen hatte, fand der optimierende Compiler bei der Modellierung keine Berücksichtigung.

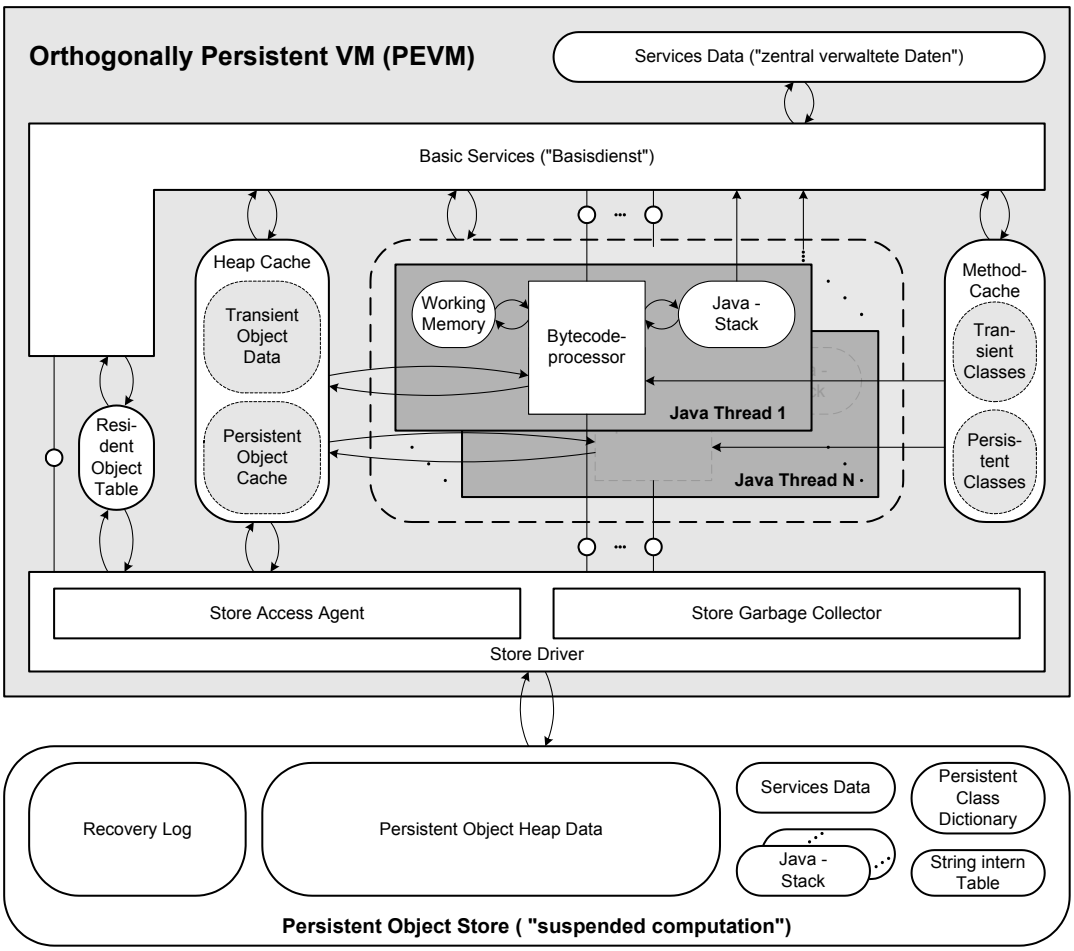


Bild 47: Orthogonally Persistent Java VM (PEVM)

cher seinerseits aus den beiden Akteuren „Store Access Agent“ und „Store Garbage Collector“ besteht. Der Store Driver ist für die Verwaltung der im Persistent Object Store abgelegten Daten zuständig. Er hat einen Kanal zu jedem der Bytecodeprozessoren, hat Zugriff auf den Heap Cache und steht mit dem Basisdienst über einen als „Resident Object Table“ bezeichneten Speicher sowie einen Kanal in Verbindung. In der Resident Object Table sind sämtliche Objekte verzeichnet, die sich aktuell im Persistent Object Cache befinden. Die Bytecodeprozessoren können die Kanäle zum Store Driver verwenden, um die Bereitstellung von Objekten anzufordern, die sich aktuell nicht im Heap Cache befinden. Die Schnittstelle zwischen Basisdienst und Store Driver wird einerseits benutzt, um bei Checkpoint Operationen die zur Rekonstruktion des VM Zustands notwendigen Daten in der „suspended computation“ abzuspeichern. Andererseits ist der Basisdienst für die Verwaltung des Heap Cache zuständig und muss aus diesem Grund mit dem Store Driver kommunizieren. Bei der Bereinigung des Heap Cache verfährt der Basisdienst nach einem Algorithmus, der eine Kombination aus einem Garbage Collection Algorithmus und einem Cache Ersetzungsalgorithmus darstellt.
Der Persistent Object Store gliedert sich in verschiedene Bereiche: Der Persistent Object Heap Bereich enthält die Daten des Java Heap, der Services Data Bereich und die Java Stack Bereiche enthalten Daten, die der Rekonstruktion der gleich benannten Speicher innerhalb der PEVM dienen, der Persistent Class Dictionary und die String Intern Table beinhalten Informationen über die von der „suspended computation“ verwendeten Java Klassen. Neben diesen fünf Bereichen beinhaltet der Persistent Object Store auch noch einen als Recovery Log bezeichneten Bereich. Dieser wird vom Store Driver benötigt, um die Atomarität von Checkpoint und von Store Garbage Collection Operationen gewährleisten zu können.

Im Fall der PEVM wird der Inhalt des Persistent Object Store prinzipiell nur im Rahmen von Checkpoint Operationen verändert. Nun müssen bei der Durchführung einer Checkpoint Operation auch die in der „suspended computation“ enthaltenen Persistent Object Heap Data so modifiziert werden, dass sie den zum Checkpoint Zeitpunkt vorliegenden Heap Inhalt widerspiegeln. Hierbei stellt sich das Problem, dass diese Operation eigentlich eine Garbage Collection des gesamten Java Heap erforderlich macht. Eine solche Garbage Collection würde die Durchführung einer Checkpoint Operation jedoch extrem zeitaufwendig machen. Um dieses Problem zu umgehen, wird bei der Durchführung einer Checkpoint Operation darauf verzichtet, Datensätze aus den Persistent Object Heap Data zu löschen, die nicht länger erreichbare Objekte repräsentieren. Die Löschung solcher Datensätze erfolgt stattdessen asynchron zu Checkpoint Operationen durch den Store Garbage Collector.

4.1.4 GemStone Facets

Auch die in diesem Abschnitt vorgestellte Entwicklung steht mit der im letzten Kapitel vorgestellten J2EE Applicationserver Technologie in engem Zusammenhang. Die Entwicklung von GemStone Facets ist durch den in Abschnitt 3.3 angesprochenen „Impedance Mismatch“ motiviert. Dieser „Impedance Mismatch“ wird insbesondere dann zum Problem, wenn mehrere J2EE Server auf die gleiche Datenbank zugreifen müssen. In Bild 48 ist diese Situation dargestellt. Am unteren Bildrand ist das relationale Datenbanksystem und die von ihm verwaltete Datenbank dargestellt. Darüber sind mehrere EJB-Container dargestellt. Jeder dieser EJB Container enthält einen Persistency Manager, der dafür zuständig ist, den EJB Side User Agents die in der relationalen Datenbank enthaltenen Daten in Form entsprechender Entity Bean Objekte zugänglich zu machen.

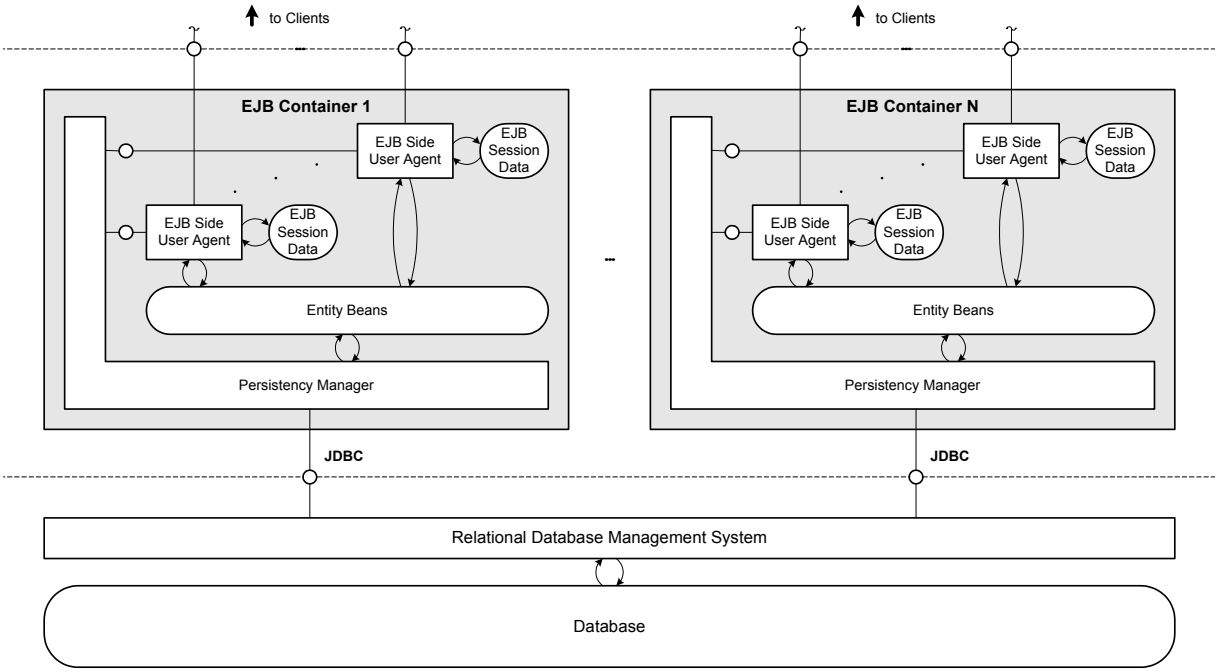


Bild 48: Nutzung einer relationalen Datenbank durch mehrere EJB Container

Der dargestellte Systemaufbau soll das grundsätzliche Problem veranschaulichen, dass Datenbankinhalte mit den in Form von Entity Beans repräsentierten Daten synchronisiert werden müssen. Wenn die Persistency Manager der unterschiedlichen EJB Container, wie im Bild dar-

gestellt, unabhäng von einander arbeiten, dann kann ein konsistenter Zugriff auf Datenbankinhalte nur indirekt über den Transaktionsmechanismus des Datenbanksystems sichergestellt werden. Dies wiederum macht die Pufferung von Datenbankinhalten durch die Persistency Manager der unterschiedlichen EJB Container praktisch unmöglich und bringt einen erheblichen Aufwand zur Erzeugung und Vernichtung von Entity Bean Objekten mit sich.

Um solche mit dem „Impedance Mismatch“ verbundenen Probleme zu vermeiden, entwickelte die Firma GemStone eine spezielle Application Server Technologie, die derzeit unter dem Namen „GemStone Facets“ vermarktet wird und nun anhand von Bild 49 näher vorgestellt werden

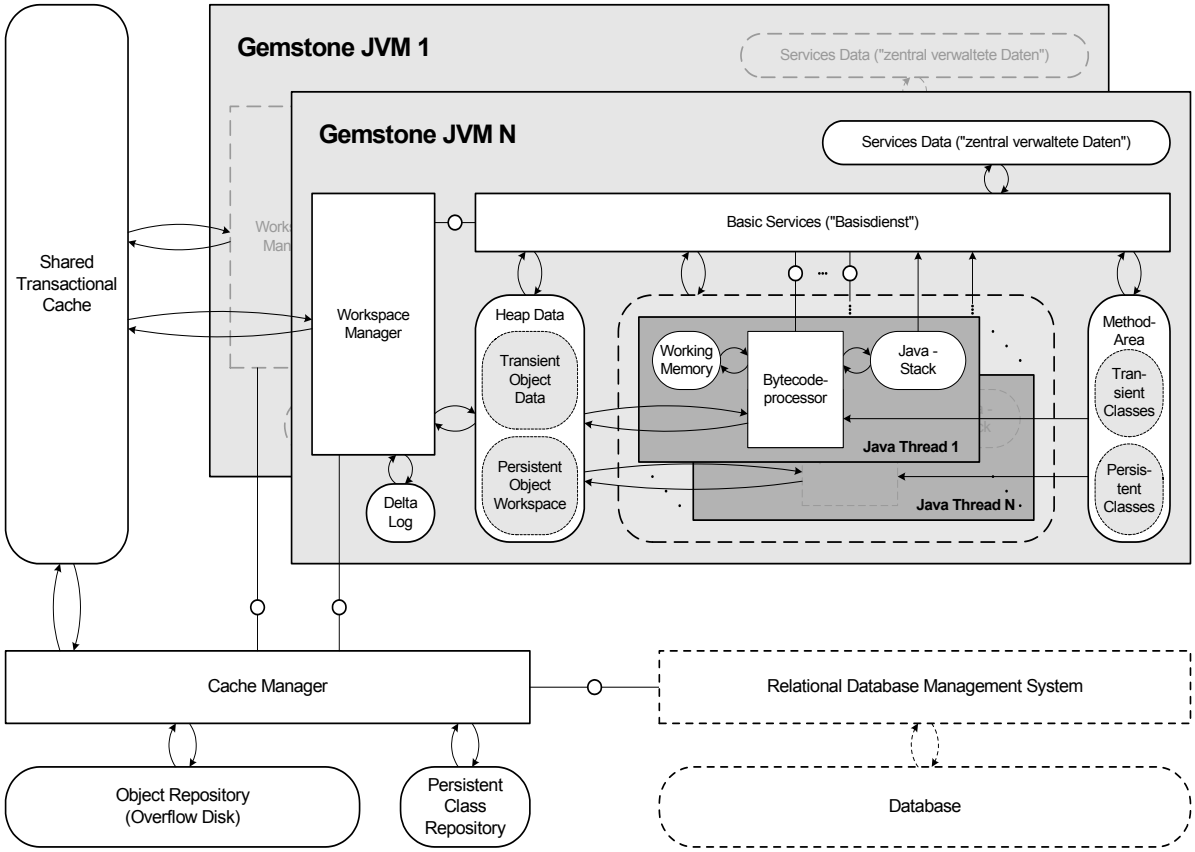


Bild 49: GemStone Facets

soll. Das Bild zeigt mehrere GemStone Java VM’s die zu einem Cluster verschaltet sind. Jede dieser VM’s hat Zugriff auf den links dargestellten Shared Transactional Cache und hat einen Kanal zu dem unterhalb der VM’s dargestellten Cache Manager. Der Cache Manager kann ebenfalls auf den Shared Transactional Cache zugreifen und hat überdies Zugriff auf die als Object Repository und Persistent Class Repository bezeichneten Speicher.
Betrachtet man die dargestellte Struktur einer GemStone Java VM genauer, so lässt sich auch hier wieder die in Abschnitt 2.2.1.1 vorgestellte Grundstruktur einer Java VM wiederfinden. Neben den zu dieser Grundstruktur gehörenden Komponenten beinhaltet eine GemStone VM zusätzlich einen Workspace Manager und einen als Delta Log bezeichneten Speicher.
Die in den verschiedenen GemStone Java VM’s enthaltenen Workspace Manager, der Cache Manager und die im Zugriff dieser Akteure liegenden Speicher stellen ein verteilt realisiertes, objektorientiertes Datenbanksystem dar. Die von diesem Datenbanksystem verwalteten Daten befinden sich im Object Repository, das Datenbankschema, welches durch die Kassendefinitionen der in der Datenbank enthaltenen Objekte gegeben ist, ist im Persistent Class Repository enthalten und der Shared Transactional Cache kann als Datenbankpuffer betrachtet werden.

Obwohl das durch die Workspace Manager und den Cache Manager realisierte Datenbanksystem bereits ein voll funktionstüchtiges Datenbanksystem darstellt, wurde im Bild noch ein weiteres, relationales Datenbanksystem angedeutet. Auf diese Weise soll auf das typische Einsatzszenario von GemStone Facets hingewiesen werden, bei dem die GemStone JVM's zur Realisierung von J2EE Servern eingesetzt werden und die durch GemStone Facets bereitgestellte objektorientierte Datenbank nur als Cache für Daten aus einem relationalen Datenbanksystem verwendet wird.

Dieses typische Einsatzszenario soll nun genauer vorgestellt werden, um den durch den Einsatz von GemStone Facets erzielbaren Vorteil zu verdeutlichen. Wie bereits angesprochen, wurde GemStone Facets entwickelt, um solche durch den „Impedance Mismatch" verursachten Probleme zu vermeiden. Die Grundidee zur Vermeidung dieser Probleme besteht darin, sämtliche zu verarbeitenden Daten vor dem Beginn der eigentlichen Verarbeitung in eine entsprechende Struktur von Objekten zu überführen, so dass auf eine (wiederholte) Konvertierung der Daten während der Verarbeitung verzichtet werden kann. Wäre die im Zugriff eines Application Servers liegende Menge an Daten nicht so umfangreich und der Bedarf an Rechenleistung für einen Application Server nicht so hoch, dann ließe sich diese Idee einfach durch eine einzelne Java VM umsetzen, die sämtliche zu verarbeitenden Daten in Form von Objekten im Java Heap bereithält. Die Konstruktion des GemStone Systems ist die konsequente Übertragung dieser Idee auf größere Systeme. Der „riesige Java Heap", der aus den Objekten besteht, welche der Repräsentation sämtlicher im Zugriff des Application Servers liegenden Daten dienen, ist in der GemStone Facets Datenbank untergebracht. Die Verarbeitung der Daten erfolgt mit Hilfe der verschiedenen GemStone Java VM's, die mit Hilfe der integrierten Workspace Manager Zugang zu diesem „riesigen Java Heap" haben. Die Workspace Manager haben dabei die Aufgabe, die in der GemStone Facets Datenbank enthaltenen Objekte auf Anfrage im Heap der betreffenden GemStone Java VM bereitzustellen und daran vorgenommene Änderungen in die GemStone Facets Datenbank zu übernehmen. Die GemStone Facets Datenbank unterstützt dabei ebenso wie relationale Datenbanksysteme das Transaktionskonzept. Ferner unterstützt die GemStone Facets Datenbank auch eine „write through cache" Betriebsart. Diese Betriebsart erlaubt es den Inhalt einer relationalen Datenbank (oder Teile davon) in der GemStone Facets Datenbank zu cachen und Änderungen, die am Inhalt dieser GemStone Facets Datenbank vorgenommen werden, automatisch mit dem Inhalt der relationalen Datenbank zu synchronisieren.

4.1.5 Software-Agenten - Mobile Code

Die Entwicklung des Konzepts der Software-Agenten war ursprünglich durch die Umgehung von Kommunikationsproblemen in Client-Server-Systemen begründet, daher wird der Agentenbegriff auch hier unter Bezug zu diesen Systemen eingeführt.
Klassischen Client-Server-Systemen liegt die Vorstellung zugrunde, dass die Clients Aufträge an den Server übermitteln. Der Server nimmt die Aufträge entgegen, bearbeitet sie und sendet die errechneten Ergebnisse als Antwort an die Clients zurück. Die Kommunikation zwischen den Clients und dem Server erfolgt dabei häufig synchron. Im Unterschied hierzu liegt Agentensystemen üblicherweise ein asynchrones Kommunikationsmodell zugrunde, welches auf der Vorstellung basiert, dass die Clients „Agenten entsenden", um die gewünschten Aufgaben zu erledigen. Zur Erfüllung ihrer Aufgabe „besuchen" die Agenten einen oder mehrere Server, um die von diesem Server angebotenen, lokalen Dienste in Anspruch zu nehmen oder um dort mit anderen Agenten zu kooperieren. Nach der Erledigung der ihnen anvertrauten Aufgabe können die Agenten schließlich wieder Kontakt zum Client aufnehmen oder zum Client zurückkehren, um gewonnene Ergebnisse abzuliefern. Obwohl prinzipiell alle Agenten mobil sind, gibt es

auch den Begriff des „mobilen Agenten“. Dieser Begriff wird normalerweise nur im Zusammenhang mit Agenten verwendet, deren Reiseroute nicht im Voraus festgelegt ist, sondern vom Agenten selbst gewählt wird. Der Prozess der Umsiedlung eines bereits vorhandenen Agenten auf einen anderen Rechner wird als Migration bezeichnet.
In der Literatur werden verschiedene Vorteile von Agentensystemen angeführt, von denen nun einige aufgezählt werden sollen: Ist ein Agent einmal auf einem Server angekommen, können zwischen dem Agenten und den übrigen auf dem Server vorhandenen Akteuren beliebig große Datenmengen ausgetauscht werden, ohne dass dabei Netzwerkverkehr entsteht. Diese lokale Kommunikation zwischen einem Agenten und den übrigen auf dem Server vorhandenen Akteuren hat gegenüber der Kommunikation über ein Netzwerk darüber hinaus den Vorteil, dass sie nicht durch lange Transportzeiten behindert wird und ist dementsprechend durch wesentlich kürzere Reaktionszeiten gekennzeichnet. Durch das asynchrone Kommunikationsmodell ist außerdem keine dauerhafte Netzwerkverbindung zwischen Clients und Servern erforderlich. Neben diesen Anwendungsfällen, die mit der Optimierung der Kommunikation zwischen Clients und Servern in Zusammenhang stehen, gibt es aber auch andere Ansätze, bei denen die Kommunikation zwischen Agenten unterschiedlicher Herkunft im Vordergrund steht. So gibt es beispielsweise auch Ansätze, um das Agentenkonzept zur Realisierung von Tauschbörsen zu verwenden. Hierbei treffen sich Agenten, die von unterschiedlichen Clients entsandt wurden, auf „elektronischen Marktplätzen“ und verhandeln dort über den Austausch von Waren.
Es ist für Agentensysteme kennzeichnend, dass die Entsendung eines Agenten an einen Server technisch so realisiert ist, dass eine aktive Systemkomponente auf Serverseite erzeugt wird, deren Verhalten vom Absender des Agenten in Form eines (imperativen/objektorientierten) Programms vorgegeben wird. Um dies zu ermöglichen, müssen auf den betreffenden Rechnern entsprechende Agentenplattformen installiert sein. Bild 50 veranschaulicht den Aufbau eines Agentensystems, es zeigt mehrere Rechner, die über das in der Bildmitte dargestellte Netzwerk verbunden sind. Auf jedem der dargestellten Rechner werden eine Agentenplattform sowie weitere (lokale) Dienste betrieben. Die innerhalb der Agentenplattformen dargestellten Agentenverwaltungsakteure haben verschiedene Aufgaben: Sie sind für Erzeugung und Zerstörung von Agenten zuständig und sorgen für eine sinnvolle Nutzung der vorhandenen Betriebsmittel durch die Agenten. Ferner bieten sie verschiedene Dienstleistungen an, um den Agenten die Nutzung der lokalen Dienste, die Kommunikation mit anderen Agenten und die Migration zu anderen Rechnern zu ermöglichen.
Die Entwicklung einer Agentenplattform steht zwangsläufig mit der Definition einer Maschine zur Ausführung von Agentenprogrammen in Zusammenhang. Da ein Agentensystem typischerweise aus einem Netzwerk heterogener Rechner besteht, ist es normalerweise nicht zweckmäßig, die konstituierenden Charakteristika solcher Maschinen aus den Eigenschaften einer bestimmten Hardware abzuleiten. Während frühe Implementierungen von Agentenplattformen auf herstellerspezifischen virtuellen Maschinen basierten, gibt es heute viele Agentenplattformen, die entweder ganz in Java implementiert sind oder speziell modifizierte Java VM’s zur Ausführung von Agentenprogrammen verwenden.

Insgesamt stellt die Idee der Software-Agenten ein sehr grundsätzliches Konzept dar, das auf unterschiedliche Weise ausgestaltet werden kann. Der Agentenbegriff ist daher nicht scharf definiert, sondern wird je nach Forschungsschwerpunkt und Sichtweise unterschiedlich verwendet.
Beispielsweise wird der Begriff auch im Kontext von Systemen verwendet, die keine Migration von Agenten unterstützen. So werden die in Abschnitt 4.1.1 vorgestellten „Java Applets“ oft als eine Sonderform von Agenten präsentiert, obwohl das Applet Konzept keinen Mechanismus zur Migration eines einmal gestarteten Applets vorsieht.
Agentenplattformen, die die Migration von Agenten unterstützen, sind aus dem Blickwinkel

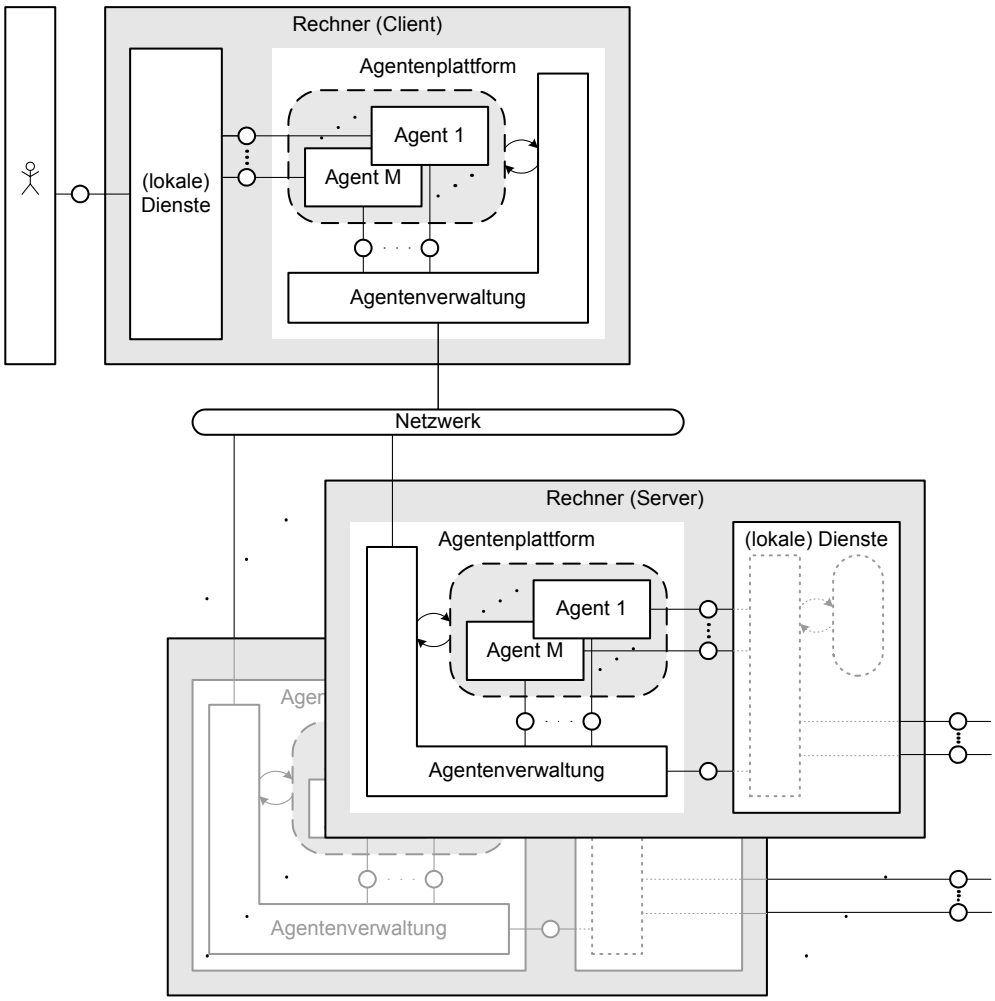


Bild 50: Agentensystem

dieser Arbeit besonders interessant, denn je nach Ausgestaltung des Migrationskonzepts stellen sie unterschiedliche Anforderungen an die virtuellen Maschinen, die zur Ausführung der Agentenprogramme verwendet werden. Hierbei sind zwei Ansätze zu unterscheiden, welche nun anhand von Bild 51 vorgestellt werden. Das Bild zeigt ein einfaches Modell einer VM zur Ausführung von Agentenprogrammen. Es besteht aus einer als Abwickler bezeichneten aktiven Komponente, welche über einen Kanal mit der Umgebung des Agenten in Verbindung steht. Im Innern des Agenten sind drei unterschiedliche Speicher dargestellt: ein Speicher für das Agen-

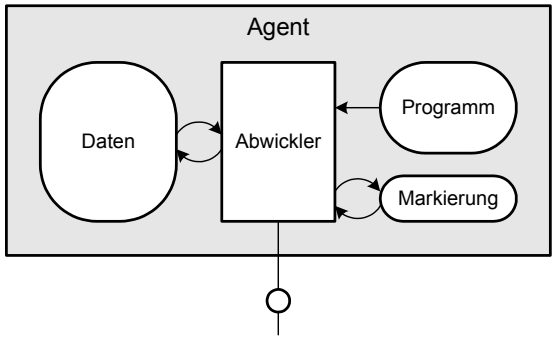


Bild 51: Einfaches Modell einer VM

tenprogramm, ein Speicher zur Ablage der mittels des Programms bearbeiteten Daten und ein

als Markierung bezeichneter Speicher, dessen Inhalt den aktuellen Fortschritt der Programmausführung wiedergibt.
Eine Möglichkeit zur Realisierung eines Migrationsmechanismus besteht darin, nur das Agentenprogramm und den aktuellen Inhalt des Datenspeichers zur Zielplattform zu übertragen und das Agentenprogramm auf dem Zielsystem neu zu starten. Da die Migration eines Agenten stets vom Agenten selbst angestoßen wird, stellt der Verlust des Markierungsspeicherinhalts prinzipiell kein Problem dar. Da der Inhalt des Markierungsspeichers in diesem Fall nicht mit übertragen wird, wird diese Form der Migration wird in der Literatur auch als „schwache Migration“ bezeichnet. Die andere, in der Literatur als „starke Migration“ bezeichnete Alternative, besteht darin, auch den Markierungszustand mit zu übertragen, so dass dieser ebenfalls auf der Zielplattform rekonstruiert und das Programm mit der auf die „Migrationsanweisung“ folgenden Anweisung fortgesetzt werden kann. Der Vorteil der starken Migration liegt in erster Linie darin, dass sie die Entwicklung des Agentenprogramms vereinfacht. Allerdings ist das Konzept der starken Migration im Allgemeinen schwieriger umzusetzen.

4.2 Problemstellungen bei der Integration

Betrachtet man die Probleme, die bei der Integration von High Level Language VMs in Systeme auftreten, so ist man zunächst einmal mit einer großen Anzahl von sehr unterschiedlichen Problemen konfrontiert, worunter es eine Vielzahl von Detailproblemen gibt, die sich einer allgemeingültigen Darstellung entziehen, weil sie nur im Kontext eines ganz bestimmten Integrationsszenarios betrachtet werden können.
Andererseits gibt es Probleme, die zwar auch bei der Integration von High Level Language VMs in Systeme in Erscheinung treten, jedoch allgemeinerer Natur sind und daher bereits in anderem Zusammenhang untersucht wurden. So kann beispielsweise bei der Gestaltung der I/O Schnittstelle einer in ein Produkt zu integrierenden virtuellen Maschine auf viele Erkenntnisse zurückgegriffen werden, die im Zusammenhang mit dem Entwurf von I/O Schnittstellen im Bereich der Betriebssysteme gewonnen wurden.
Betrachtet man die in der Literatur beschriebenen Integrationsszenarien, so ist festzustellen, dass die Integration von virtuellen Maschinen in bestehende Systeme häufig in der Absicht geschieht, eine Schnittstelle für Systemerweiterungen zu schaffen. Ferner ist zu beobachten, dass bei der Integration von virtuellen Maschinen in bestehende Systeme häufig nicht genau eine virtuelle Maschine integriert wird, sondern dass meist mehrere (gleiche) VMs in das betreffende System integriert werden.
Dies belegen auch die bereits vorgestellten Beispiele: Im Fall der Integration der Java VM in den SAP Application Server wurde jedem Benutzer eine eigene Java VM zur Verfügung gestellt, um die Abschottung der Sitzungen unterschiedlicher Benutzer gewährleisten zu können. Datenbanksysteme nutzen zur Ausführung von „stored procedures“, die in Form von Java Klassen formuliert sind, ebenfalls mehrere Java VMs. Hier ist die Nutzung mehrerer Java VMs notwendig, um die „Isolation“[32] verschiedener Transaktionen garantieren zu können. Einige Agentenplattformen, bei denen die Agentenprogramme in Form von Java Klassen formuliert werden, stellen für jeden Agenten eine eigene Java VM bereit. Diese 1:1 Abbildung zwischen Agenten und Java VMs vereinfacht es erheblich, den Betriebsmittelbedarf einzelner Agenten festzustellen und „außer Kontrolle“ geratene Agenten zu entfernen.
Der Grund für die Nutzung mehrerer VMs liegt in allen diesen Fällen darin, dass die Java VM primär nicht zum Betrieb mehrerer logisch separater Anwendungen konzipiert wurde und daher

32. „ACID“ Eigenschaften

kein brauchbares „Prozess“ Konzept anbietet, welches es ermöglichen würde, mehrere logisch separate Anwendungen innerhalb einer Java VM gegeneinander abzugrenzen. Das Fehlen eines brauchbaren „Prozess“ Konzepts ist dabei kein Java VM spezifisches Problem, sondern für High Level Language VMs geradezu typisch.
Aufgrund der besonderen Praxisrelevanz dieses Problems, steht es im Vordergrund der weiteren Betrachtungen. Dabei ist zunächst einmal festzustellen, dass die Nutzung mehrerer Java VMs das Fehlen eines brauchbaren Prozess Konzepts nur bedingt ausgleichen kann, weil dadurch ein erheblicher Zusatzaufwand verursacht wird. Einerseits ist das Booten einer Java VM mit erheblichen Rechenzeitaufwand verbunden. Andererseits zieht der Betrieb mehrerer Java VMs einen großen Hauptspeicherbedarf nach sich. Der durch das Fehlen eines brauchbaren Prozess Konzepts verursachte Zusatzaufwand wird im Folgenden als Startupaufwand bezeichnet und kann in erster Näherung durch die nachstehende Formel charakterisiert werden:

$$\underline{\text{Startupaufwand}} = \underline{\text{Einzel-Startupaufwand}} * \underline{\text{Startupbedarf pro Zeit}}$$

Die Formel soll auf zwei unterschiedliche Lösungswege zur Reduktion des Startupaufwands hinweisen. Einerseits kann man versuchen, den Aufwand zum Start einer einzelnen Java VM („Einzel-Startupaufwand“) zu reduzieren. Die andere Möglichkeit besteht darin, durch Einführung eines Prozess Konzepts für eine klare Abgrenzung verschiedener Anwendungen innerhalb einer Java VM zu sorgen und so die Anzahl der insgesamt benötigten Java VMs zu reduzieren. In den folgenden beiden Abschnitten 4.2.1 und 4.2.2 werden verschiedene in der Literatur beschriebene Systeme vorgestellt. Die Darstellung in Abschnitt 4.2.1 beschäftigt sich mit dem Problem der Reduktion des „Einzel-Startupaufwands“. In Abschnitt 4.2.2 wird das Problem der Abgrenzung verschiedener Anwendungen innerhalb einer Java VM thematisiert.

4.2.1 Reduktion des Startupaufwands virtueller Maschinen

In diesem Abschnitt soll näher darauf eingegangen werden, wie der zum Betrieb mehrerer Java VM’s notwendige Betriebsmittelbedarf reduziert werden kann, wenn diese VM’s auf dem gleichen Rechner betrieben werden. Den Ausgangspunkt für die Entwicklung dieser VM’s bildeten zwei Feststellungen. Die erste Feststellung ist die, dass ein beträchtlicher Teil der Betriebsmittel, die für den Betrieb einer Java VM notwendig sind, zur Aufbereitung und Speicherung des auszuführenden Codes aufgewendet werden. Die zweite Feststellung ist die, dass der gleiche Bytecode häufig von mehreren VM’s benötigt wird. Ein gewisses Maß an Überdeckung des von verschiedenen Java VM’s verwendeten Bytecodes wird dabei durch die „Standard Class Libraries“ verursacht, da praktisch jedes Java Programm die darin implementierte Funktionalität nutzt. Speziell wenn Java im Serverbereich eingesetzt wird, beispielsweise zum Betrieb mehrerer J2EE Application Server, liegt häufig ein noch größerer Überdeckungsgrad des von verschiedenen Java VM’s verwendeten Bytecodes vor.

„Code Sharing“ zwischen Virtuellen Maschinen

Bei herkömmlichen Java VM Implementierungen ist jede Java VM durch einen völlig eigenständigen Betriebssystem Prozess realisiert. Demzufolge muss jede Java VM sämtlichen auszuführenden Bytecode aufbereiten und in einer für die spätere Ausführung geeigneten Form im Hauptspeicher bereithalten. Daher entsteht in diesem Fall für Bytecode, der von mehreren Java VM’s verwendet wird, mehrfach Aufbereitungs- und Speicheraufwand. In diesem Abschnitt werden nun VM’s vorgestellt, mit denen das Ziel verfolgt wird, den normalerweise mehrfach auftretenden Aufbereitungs- und Speicheraufwand für Bytecode, der von mehreren Java VM’s verwendet wird, zu reduzieren.

Zunächst werden drei verschiedene, alternative Lösungsansätze der Firma Sun vorgestellt, die sämtlich mit der Multitasking Virtual Machine in Zusammenhang stehen (siehe [Czajkowski Daynès 01], [Czajkowski et al 01-NCI], [Czajkowski et al 02], [Wong et al. 03]). Um zu vergleichbaren Ergebnissen zu gelangen, nutzten die Forscher der Firma Sun in allen drei Fällen die gleiche, dem aktuellen Stand der Technik entsprechende HotSpot Virtual Machine als Ausgangspunkt (siehe [Sun-HSVM]). Zuerst wird die Multitasking Virtual Machine vorgestellt, welche der vielversprechendste dieser Ansätze ist. Ihre Konstruktion basiert darauf, die Ausführung des Java Bytecodes mehrerer VM's in einem einzigen Betriebssystemprozess zu „zentralisieren". Der zweite Ansatz basiert auf der Nutzung von „shared memory". Im dritten Fall wird das gewünschte Sharing mittels „shared libraries" realisiert.

„Wiederverwendung" virtueller Maschinen

Im Anschluß an die Vorstellung der verschiedenen Varianten der MVM werden einige Forschungsaktivitäten der Firma IBM vorgestellt. In diesem Zusammenhang wird auch auf eine Java VM Implementierung hingewiesen, welche speziell im Hinblick auf Transaktionsverarbeitende Systeme entworfen wurde. Ihrer Konzeption liegt die Idee zugrunde, den Startupaufwand durch „Wiederverwendung" von VMs zu reduzieren.

4.2.1.1 „Multitasking Virtual Machine"

Der Aufbau der Multitasking Virtual Machine soll nun anhand der Bilder 52 und 53 näher vorgestellt werden. Bild 52 zeigt die übliche Abbildung von Java VM's auf Betriebssystemprozes-

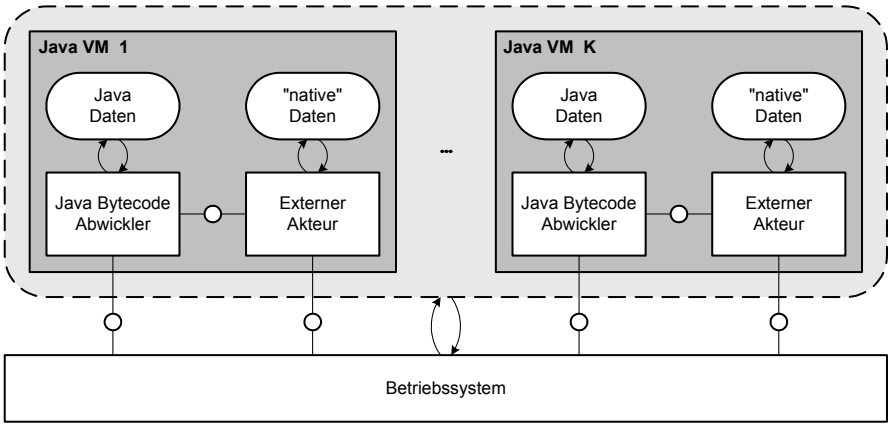


Bild 52: Herkömmliche 1:1 Abbildung von Java VM's auf Betriebssystemprozesse

se. Im Bild sind mehrere herkömmliche Java VM Prozesse angedeutet, von denen genau zwei dargestellt sind. Sie dienen jeweils der Realisierung einer einzigen Java VM. Die im Innern dieser Prozesse dargestellte Struktur wurde bereits anhand von Bild 13 in Abschnitt 2.2.1.2 eingeführt und besteht aus einem Akteur, der für die Auswertung von Java Bytecode zuständig ist und einem zweiten Akteur, der diejenigen Aufgaben übernimmt, die mit Hilfe von „native" Methoden implementiert sind.

Bild 53 zeigt die im Fall der MVM vorliegende Abbildung zwischen Java VM's und Betriebssystemprozessen. Wie bereits erwähnt, basiert die Konzeption der Multitasking VM darauf, dass die Ausführung des Java Bytecodes sämtlicher VM's in einem einzigen Betriebssystemprozess „zentralisiert" wird. Dieser Betriebssystemprozess ist im Bild in Form des „j-process" wiederzufinden. Die in seinem Innern dargestellte Struktur soll den Zusammenhang zu Bild 52

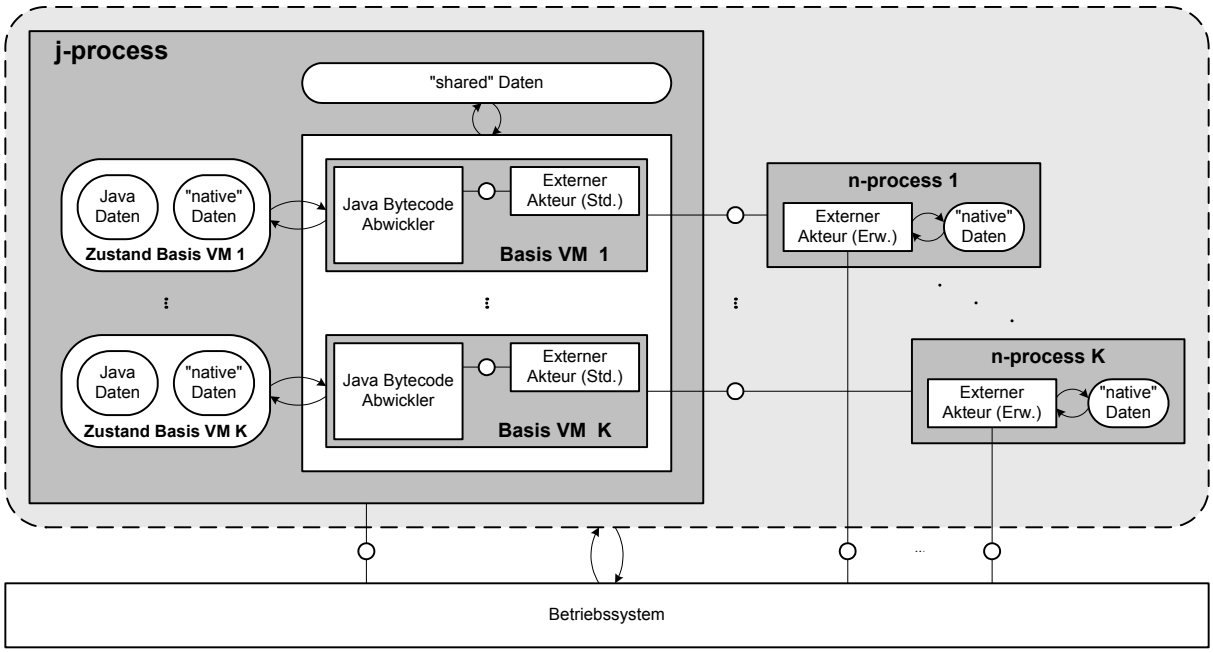


Bild 53: Multitasking Virtual Machine

herstellen. Für jede zu realisierende Java VM ist im Inneren des „j-process“ ein Basis VM Akteur dargestellt. Weiterhin ist jedem Basis VM Akteur ein Speicher zugeordnet, in dem der für die betreffende VM spezifische Teil des VM Zustands gespeichert ist. Des Weiteren haben alle Basis VM Akteure Zugriff auf einen Speicher für „shared“ Daten. Er bildet die Grundlage für das Codesharing zwischen den verschiedenen, auf Basis der MVM realisierten Java VM’s. Das Codesharing ist im Fall der MVM zweigestaltig, da die MVM intern zwei verschiedene Darstellungen des auszuführenden Codes benutzt. Einerseits wird sämtlicher Java Code intern in einer Bytecode ähnlichen Form repräsentiert. Andererseits beinhaltet die MVM auch einen optimierenden Compiler der plattformspezifischen Maschinencode erzeugt. Im Fall der MVM wird sowohl die Bytecodedarstellung als auch der vom Compiler erzeugte Maschinencode im „shared“ Daten Bereich abgelegt und von allen Java VM’s gemeinsam benutzt. Im Unterschied zu Bild 52 gibt es in Bild 53 je zwei Externe Akteure pro Java VM, die überdies in unterschiedlichen Betriebssystemprozessen beherbergt sind. Der Grund hierfür liegt darin, dass die Funktionalität des Externen Akteurs einer auf Basis der MVM realisierten Java VM erweitert werden kann. Bei einer herkömmlichen Java VM würde der Maschinencode, der zur Erweiterung der Funktionalität des Externen Akteurs dient, einfach in den Adressraum des Java VM Prozesses geladen werden. Im Fall der MVM ist die Funktionalität des in der Basis VM untergebrachten Externen Akteurs durch die Konstruktion der MVM fest vorgegeben. Der durch ihn realisierte Funktionsumfang entspricht dabei im Wesentlichen demjenigen, der für die Standard Class Libraries erforderlich ist (J2SE). Wenn die Funktionalität des Externen Akteurs einer bestimmten Java VM erweitert werden soll, erzeugt die MVM einen separaten, der betreffenden Java VM zugeordneten „n-process“ und lädt den betreffenden Code in den Adressraum dieses Betriebssystemprozesses.
Diese Form der Realisierung bringt den Nachteil mit sich, dass die Kommunikation zwischen einer Basis VM und dem ihr zugeordneten, in den „n-process“ ausgelagerten Externen Akteur mit größerem Aufwand verbunden ist als dies bei einer herkömmlichen Java VM der Fall wäre. Diesem Nachteil, der überhaupt nur dann in Erscheinung tritt, wenn die Funktionalität des Externen Akteurs tatsächlich erweitert wird, stehen jedoch große Vorteile gegenüber:

- Einerseits erlaubt diese Form der Realisierung die Unterstützung des **J**ava **N**ative **I**nterface (JNI) durch die MVM. Aus diesem Grund können Programmbibliotheken, die der Erweiterung der Funktionalität des Externen Akteurs dienen und hierzu das JNI nutzen, sowohl für auf herkömmliche Weise realisierte Java VM's als auch für solche Java VM's verwendet werden, die auf Basis der MVM realisiert sind.
- Andererseits kann die wechselseitige Abschottung der verschiedenen auf Basis der MVM realisierten Java VM's auch dann garantiert werden, wenn die zur Erweiterung der Funktionalität des Externen Akteurs dienenden „native" Code Bibliotheken fehlerhaft sind.

4.2.1.2 „shared memory" Multitasking Virtual Machine

Bild 54 zeigt eine Variante der Multitasking Virtual Machine, bei der das Java VM übergreifende Sharing von Daten mit Hilfe eines „shared memory" Bereiches realisiert wird. Sie wird in Anlehnung an die Originalveröffentlichung im Folgenden als ShMVM bezeichnet. Ganz oben in Bild 54 ist der von den verschiedenen Java VM's gemeinsam genutzte Speicher für „shared

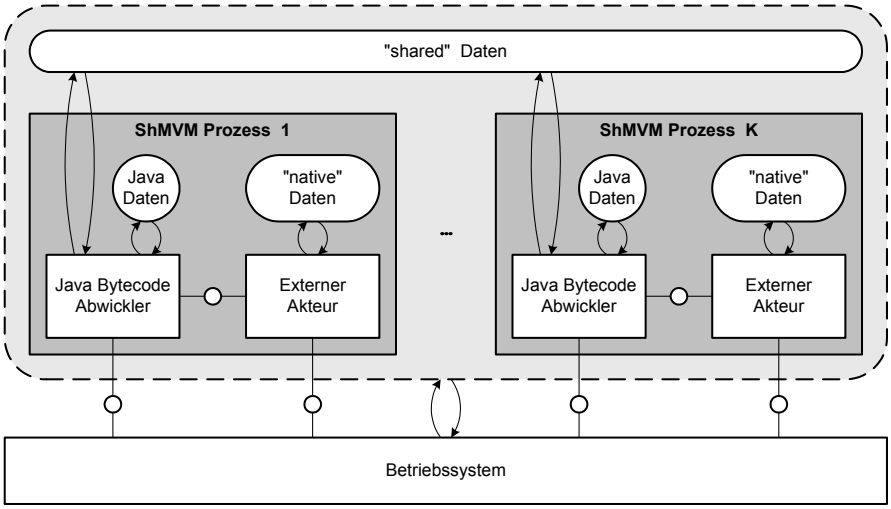


Bild 54: „shared memory" Multitasking Virtual Machine

Daten dargestellt. Darunter sind die verschiedenen ShMVM Betriebssystemprozesse dargestellt, die jeweils der Realisierung einer einzelnen Java dienen. Ebenso wie bei der MVM, wurde auch bei der ShMVM eine Java VM übergreifende Nutzung der Bytecodedarstellung und des von den optimierenden Compilern erzeugten Maschinencodes realisiert. Die verschiedenen ShMVM Prozesse sind bezüglich der Nutzung der „shared Daten" gleichberechtigt. Auch die ShMVM unterstützt das JNI. Im Gegensatz zur MVM wird der Maschinencode, der zur Erweiterung der Funktionalität des Externen Akteurs einer Java VM dient, direkt in den Adressraum des betreffenden ShMVM Prozesses geladen. Dadurch wird die bei der MVM herrschende Aufteilung der zur Realisierung einer Java VM dienenden Komponenten auf mehrere Betriebssystemprozesse sowie der damit verbundene Zusatzaufwand vermieden. Die Konzeption der ShMVM bringt im Unterschied zur MVM das Problem mit sich, dass die wechselseitige Abschottung der verschiedenen auf Basis der ShMVM realisierten Java VM's nicht durch die Konstruktion der ShMVM sichergestellt werden kann. Programmierfehler in dem zur Erweiterung der Funktionalität des Externen Akteurs einer Java VM dienenden Maschinencode können prinzipiell dazu führen, dass der Inhalt des „shared Daten" Bereichs unbrauchbar wird. Nun könnte man versuchen, die im „shared Daten" Bereich abgelegten Daten vor ungewollten Änderungen

zu schützen, indem man diesen Speicherbereich nur während der Durchführung von „gewollten“ Änderungen „read write“ und ansonsten nur „read only“ in die Adressräume der ShMVM Prozesse einblendet. Im Fall der ShMVM ist es jedoch fragwürdig, ob die „shared Daten“ auf diese Weise sinnvoll geschützt werden können. Hierbei muss Folgendes bedacht werden:

- Der Adressraum eines ShMVM Prozesses wird von mehreren, potentiell nebenläufigen Betriebssystemthreads „benutzt“. Dies ist einerseits dadurch gegeben, dass die ShMVM Implementierung selbst mehrere Betriebssystemthreads benutzt. Andererseits ist es durchaus zulässig, dass im Verlauf der Abwicklung des Maschinencodes, der zur Erweiterung der Funktionalität des Externen Akteurs dient, zusätzliche Betriebssystemthreads erzeugt werden. Deshalb kann nicht ausgeschlossen werden, dass solche Teile des „shared Daten“ Bereichs, bei denen der Schreibschutz vorübergehend aufgehoben wurde, weil ein Thread reguläre Veränderungen an deren Inhalt durchführen muss, nicht zeitgleich durch die übrigen Threads unbrauchbar gemacht werden.
- Die ShMVM enthält einen optimierenden Compiler. Während der Ausführung von Java Programmteilen werden entsprechende Statistiken erstellt, die als Entscheidungsgrundlage für die Triggerung von Übersetzungsvorgängen und die Steuerung von Optimierungen verwendet werden. Da mit der ShMVM insbesondere auch eine Java VM übergreifende Verwaltung/Nutzung des von den optimierenden Compilern erzeugten Maschinencodes angestrebt wird, ist es erforderlich, dass auch die zur Steuerung von Optimierungen dienenden Statistiken allen VM’s zugänglich sind. Sie sind daher ebenfalls im „shared Daten“ Bereich untergebracht. Da diese Statistikdaten ständig aktualisiert werden, würde es einen nicht vertretbaren Aufwand erfordern, die Speicherschutzeinstellungen der zur Ablage dieser Statistikdaten dienenden Speicherbereiche fortwährend zu ändern. Aufgrund der Tatsache, dass die Implementierung der ShMVM auf der Implementierung der HotSpot VM basiert, sind diese Statistikdaten vom Speicherlayout her nicht klar von den übrigen Daten im „shared Daten“ Bereich separiert.

Aufgrund der gerade geschilderten Zusammenhänge wurde bei der Implementierung der ShMVM auf einen Schutz des „shared Daten“ Bereichs gänzlich verzichtet. Obwohl die Konstruktion der ShMVM den beschriebenen Nachteil aufweist, bietet sie möglicherweise Vorteile, wenn die verschiedenen Java VMs bei der Benutzung der Betriebssystemschnittstelle unterschiedliche User-IDs verwenden sollen.

4.2.1.3 Verwendung von „shared libraries“

Im Fall der ShMVM wurde die aus dem Bytecode abgeleitete VM interne Darstellung zum Zwecke des Sharings in einem „shared memory“ Bereich untergebracht, der allen ShMVM Prozessen zugänglich war. Im Unterschied hierzu nutzt die in [Wong et al. 03] beschriebene SLMVM „shared libraries“ um ein Betriebssystemprozess-übergreifendes Sharing zu implementieren. Hierzu stellt das System einen „Übersetzer“ zur Erzeugung entsprechender „shared libraries“ bereit. Dieser Übersetzer bekommt als Eingabe eine bestimmten Menge von Java-Klassen. Im Verlauf der Übersetzung überführt er diese Klassendefinitionen in eine Form, die der internen Darstellung von Klassen während des Betriebs einer SLMVM entspricht und „speichert“ sie in der erzeugten „shared library“ ab. Wie bei „herkömmlichen“ Java VM Implementierungen ist jede unter Verwendung der SLMVM Implementierung hergestellte Java VM als eigenständiger Betriebssystemprozess realisiert. Der Unterschied zu „herkömmlichen“ Java VM Implementierungen besteht lediglich darin, dass alle diese SLMVM Prozesse die zuvor erzeugten „shared libraries“ gemeinsam Nutzen.

4.2.1.4 „serial reuse“ von VM’s

Auch die Firma IBM hat das Thema der Reduktion des Startupaufwands intensiv erforscht. Neben den Ansätzen die von der Firma Sun im Kontext der Entwicklung der MVM untersucht wurden, hat die Firma IBM auch die Möglichkeit der Wiederverwendung von VMs untersucht. Die Idee zur Wiederverwendung von VMs ist naheliegend, wenn man die Art der Nutzung von Java VMs durch Datenbanksysteme betrachtet. Um die „Isolation“ der verschiedenen von einem Datenbanksystem bearbeiteten Transaktionen zu wahren, ist es praktisch unumgänglich, für jede Datenbanktransaktion, in der die Ausführung einer „stored procedure“ verlangt wird, eine „neue“ Java VM zu verwenden. Entsprechende Untersuchungen zu diesem Thema haben ergeben, dass der Aufwand zum Start einer Standard Java VM typischerweise um 1 bis 2 Größenordnungen höher ist als derjenige Aufwand, der für die innerhalb einer Transaktion auszuwertenden „stored procedures“ erforderlich ist.
Die von IBM angestellten Untersuchungen haben zur Entwicklung einer entsprechenden VM geführt. Informationen zu dieser VM sind in den beiden Veröffentlichungen [Dillenberger et al. 00] und [Borman et al. 01] zu finden. Da diese beiden Veröffentlichungen selbst nicht sehr detailliert auf die technische Umsetzung dieser VM eingehen, muss die nun folgende Darstellung zwangsläufig ebenfalls etwas unpräzise ausfallen. Bild 55 zeigt den Grund-

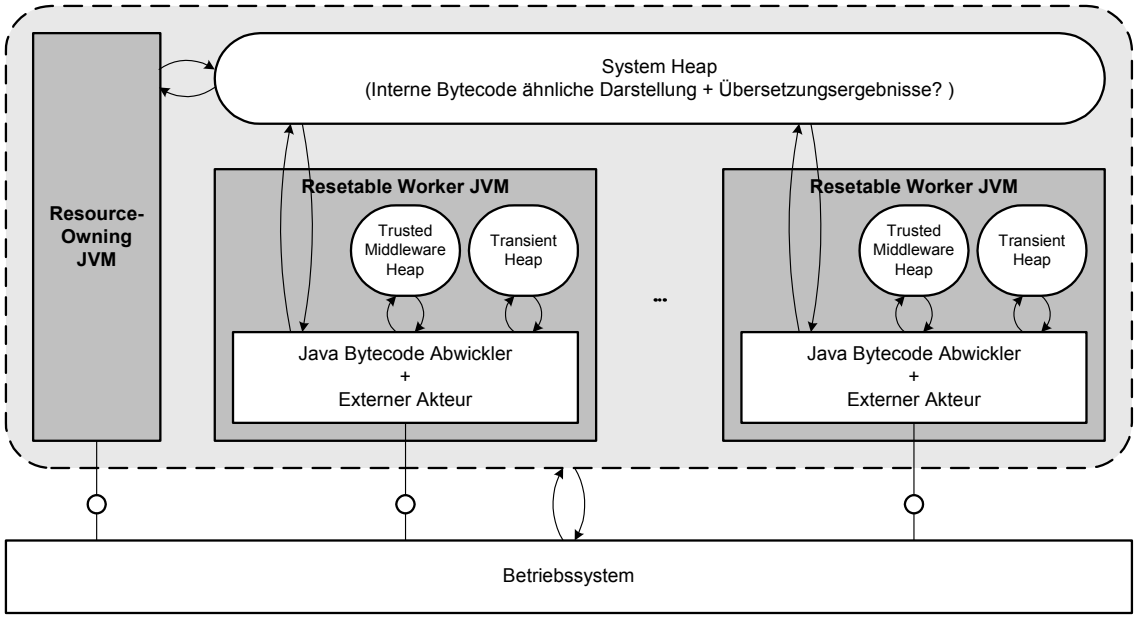


Bild 55: „Serially Reuseable Java VM“

sätzlichen Aufbau der „Serially Reuseable Java VM“, die gelegentlich auch als „Resetable Java VM“ bezeichnet wird. Das System besteht aus mehreren Prozessen, die über einen als „System Heap“ bezeichneten „shared memory“ Bereich miteinander verkoppelt sind. Jeder dieser Prozesse dient der Realisierung einer eigenständigen Java VM. Einer der Prozesse wird als „Resource-Owning JVM“ bezeichnet und hat eine nicht näher beschriebene Sonderrolle. Neben diesem „Resource-Owning JVM“ Prozess gibt es eine variable Anzahl an „Resettable Worker JVM“ Prozessen. Wie der Name schon andeutet, sind die „Resetable Worker JVM“ Prozesse diejenigen, die für die Ausführung der im Rahmen der Transaktionsverarbeitung auszuwertenden „stored procedures“ vorgesehen sind. Diese VMs haben die Eigenschaft, dass sie nach Abschluss einer Transaktion wieder in einen definierten „Initialzustand“ gebracht werden können. Der als „System Heap“ bezeichnete „shared memory“ Bereich bildet die Grundlage für das VM übergreifende Sharing von Bytecode und der daraus erzeugten internen Darstellungen. Da sämt-

liche VMs das „Java Native Interface“ unterstützen, stellt sich die Frage, ob und wie die Integrität des System Heaps sichergestellt wird. Diese Frage kann aber mangels Information nicht geklärt werden. Das System erlaubt es einerseits mit geringem Aufwand neue „Resetable Worker JVM“ Prozesse zu erzeugen. Andererseits sieht es die Möglichkeit vor, einmal erzeugte „Resetable Worker JVM“ Prozesse (mit noch geringerem Aufwand) in einen definierten Grundzustand zurückzuversetzen, so dass sie mehrfach verwendet werden können. Jede „Resetable Worker JVM“ verfügt über je einen eigenen „Trusted Middleware Heap“ und „Transient Heap“ zur Ablage privater Daten. Diese Zweiteilung resultiert aus dem Wunsch, bei einem VM-Reset auf eine vollständige Garbage Collection verzichten zu können. Ein solcher VM-Reset erfolgt daher in zwei Schritten: Im ersten Schritt wird der Versuch unternommen, den „Trusted Middleware Heap“ „aufzuräumen“. Wenn dieser Versuch glückt, wird der Inhalt des „Transient Heap“ einfach gelöscht. Scheitert dieser Versuch, so wird der gesamte „Resetable Worker JVM“ Prozess beendet.

4.2.2 Integration des Prozess Konzepts in High Level Language VM’s

Java VM’s werden seit langer Zeit dazu benutzt, gleichzeitig mehrere verschiedene logisch separate „Anwendungen“ auszuführen. Ein sehr bekanntes Beispiel für eine derartige Nutzung der Java VM stammt aus dem Jahre 1995 und wurde bereits in Abschnitt 4.1.1 vorgestellt. Dabei handelt es sich um das Web-Browser Beispiel, bei dem eine Java VM zur Ausführung mehrerer Applets eingesetzt wird. Da sich die begrenzte Eignung der Java VM zur Ausführung unterschiedlicher „Anwendungen“ bereits damals zeigte, wurde die hierfür angebotene Unterstützung in der weiteren Folge ausgebaut (siehe [Balfanz Gong 97],[Gong 99]). Dennoch bietet die Java VM bis heute kein Konzept an, das mit dem aus der Betriebssystemwelt bekannten „Prozess“ Konzept vergleichbar ist. Im gerade zitierten Web-Browser Beispiel zeigt sich dies beispielsweise darin, dass es unmöglich ist, die Menge der zum Betrieb eines bestimmen Applets aufgewendeten Betriebsmittel festzustellen.
Die Idee des in klassischen Betriebssystemen implementierten Prozess Konzepts besteht darin, mit Hilfe eines Rechners mehrere virtuelle Maschinen zu realisieren. Die seitens des Betriebssystems geleistete Unterstützung zur Bereitstellung solcher „Process VMs“ hat dabei verschiedene Aspekte:

- Abschottung
  Der Zustand, des durch einen Prozess realisierten Akteurs, ist von dem der übrigen Akteure klar abgegrenzt, da er, sofern kein „shared memory“ benutzt wird, in einem separaten Adressraum abgelegt ist.
- Betriebsmittelverwaltung
  Das Betriebssystem führt Buch über die zum Betrieb der verschiedenen Prozesse verwendeten Betriebsmittel und überwacht die Zuteilung von Betriebsmitteln an Prozesse.
- Kommunikation
  Da ein System im Allgemeinen unter Verwendung mehrerer kooperierender Prozesse realisiert ist, muss das Betriebssystem effiziente Mechanismen zur Kommunikation der auf Basis unterschiedlicher Prozesse realisierten Akteure bereitstellen.

Nun unterscheidet sich das Maschinenmodell einer modernen High-Level-Language VM wie der Java VM grundsätzlich von dem einer klassischen ISA VM. Ein zentraler Unterschied zwischen den beiden Maschinenmodellen liegt darin, dass die Java VM die Einhaltung des typgerechten Zugriffs auf Daten überwacht. Andererseits ist der Begriff des linearen Adressraums, der untrennbar mit dem klassischen Prozess-Begriff verbunden ist, für die Definition des Ma-

schinenmodells der Java VM ohne jede Relevanz. Aufgrund dieser Unterschiede zwischen den Maschinenmodellen gibt es auch wesentliche Unterschiede bei den zur Realisierung von Prozess Konzepten verwendeten Techniken.

Multitasking auf Basis mehrerer „Standard“ Java VM’s

Bevor die Frage diskutiert wird, wie mehrere „Java Process VM’s“ innerhalb einer „Multi-Prozess“ Java VM gegeneinander abgegrenzt werden können, sollen die Gründe vorgestellt werden, warum solche Systeme, bei denen eine Abgrenzung verschiedener Systemkomponenten auf Basis von „Java Process VM’s“ sinnvoll ist, nicht ebenso gut dadurch hergestellt werden können, dass jede Systemkomponente auf Basis einer eigenen „Standard“ Java VM realisiert wird.

- Hoher Betriebsmittelbedarf
  Der Bedarf an Betriebsmitteln zum Betrieb mehrerer „Standard“ Java VM’s ist normalerweise höher als derjenige, der zum Betrieb einer einzigen „Multi-Prozess“ Java VM notwendig ist. Eine Ursache hierfür ist beispielsweise, dass das in Abschnitt 4.2.1 beschriebene „Code Sharing“ in „Standard“ Java VM Implementierungen nicht umgesetzt ist.
- Keine „Java Schnittstelle“ zur Verwaltung verschiedener Java VM’s
  Die Schnittstellen zur Erzeugung von „Standard“ Java VM’s sowie zur Überwachung der zum Betrieb einzelner Java VM’s aufgewendeten Betriebsmittel sind betriebssystemabhängig und entziehen sich daher der Nutzung durch Java Programme.
- Hoher Kommunikationsaufwand
  Wenn die einzelnen Komponenten eines Systems auf Basis von „Standard“ Java VM’s realisiert werden, kann die Kommunikation zwischen den verschiedenen Systemkomponenten nur sehr ineffizient realisiert werden.

Java Application Isolation API

In der Vergangenheit gab es bereits vielfältige Bemühungen zur Erweiterung der Java VM um ein Prozess Konzept, das mit dem aus der Betriebssystemwelt bekannten vergleichbar ist. Inzwischen ist aus diesen Bemühungen ein Standard Java API zur Verwaltung von Java „Prozessen“ hervorgegangen [JSR-121-03]. Die durch diesen Standard definierten Begriffe sollen nun anhand von Bild 56 vorgestellt werden. In der Begriffswelt dieses Standards werden Java Prozesse als Isolates bezeichnet. Zwei solche Isolates sind in der linken Bildhälfte dargestellt. Der Standard verlangt, dass jedes Isolate sich aus Sicht eines Java Programmierers wie eine völlig eigenständige Java VM verhält.[33] Der Standard lässt verschiedene Formen der Realisierung von Isolates zu. Insbesondere lässt der Standard offen, ob die verschiedenen Isolates auf Basis eines einzigen Betriebssystemprozesses realisiert sind, ob jedes Isolate durch einen separaten Betriebssystemprozess realisiert ist, oder ob die verschiedenen Isolates über mehrere Rechner verteilt sind. Dem Isolate API liegt die Vorstellung zugrunde, dass die Implementierung einer „Multi-Isolate-Java-VM“ so gestaltet ist, dass sie als Ersatz für Standard Java VM Implementierungen dienen kann. Demzufolge würde eine solche „Multi-Isolate-Java-VM“ beim Start zunächst nur ein „primordial Isolate“ erzeugen und dieses zur Ausführung der „main()“-Methode der beim Start angegebenen Klasse verwenden. Im Verlauf der Programmausführung kann dieses „primordial Isolate“ das Isolate-API nutzen, um weitere Isolates zu erzeugen. Die Erzeugung von Isolates ist dabei nicht nur dem „primordial Isolate“ vorbehalten, sondern kann

33. Der Standard macht Einschränkungen bezüglich der wechselseitigen Abschottung der VM’s, wenn das Java Native Interface genutzt wird.

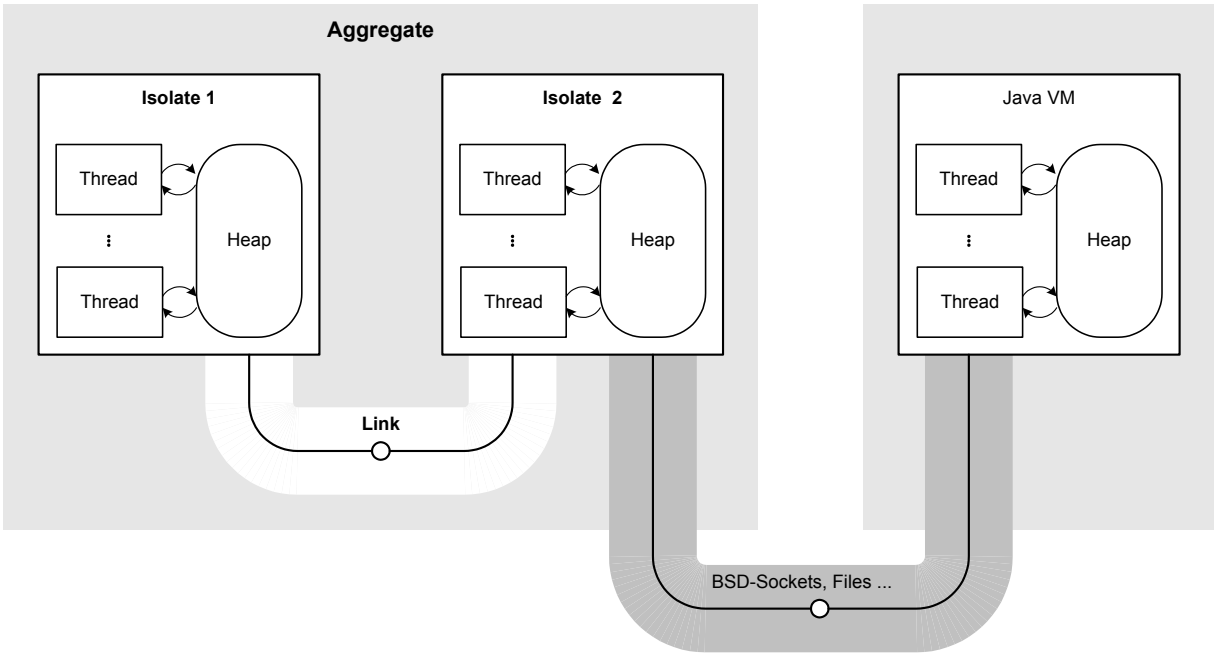


Bild 56: „Java Application Isolation API“ - Begriffe

prinzipiell von jedem Isolate angestoßen werden. Sämtliche ausgehend von einem „primordial Isolate“ erzeugten Isolates bilden ein System von Isolates, welches in der Terminologie der Spezifikation als Aggregate bezeichnet wird. Wie im vorigen Abschnitt beschrieben, ist die Kommunikation zwischen verschiedenen Java VM’s im Allgemeinen mit großem Aufwand verbunden. Um die Implementierung effizienterer Kommunikationsmechanismen zwischen verschiedenen Isolates eines Aggregate zu ermöglichen, sieht der Standard das Konzept der Links vor. Links sind dabei Punkt zu Punkt Verbindungen über die, ähnlich wie über TCP-Verbindungen, Nachrichten ausgetauscht werden. Während über TCP-Verbindungen nur Bytefolgen übertragen werden können, erlauben Links den Austausch von Nachrichten, die in Form einer Objektstruktur vorliegen. Der Versand einer Nachricht über einen Link hat aus Sicht eines Java Programmierers „Kopiersemantik“[34]. Das heißt, dass eine Kopie der Objektstruktur, die die zu versendende Nachricht repräsentiert, im Heap des Empfängers angelegt wird. Ein direktes „sharing“ von Objekten zwischen verschiedenen Java VM’s sieht der Standard hingegen nicht vor.
Eine Effizienzsteigerung bei der Kommunikation über Links im Vergleich zur Kommunikation über andere Kommunikationsmechanismen ist aus verschiedenen Gründen möglich. Zur Diskussion dieser Gründe sind in Bild 57 die verschiedenen Schritte dargestellt, die bei der Nutzung herkömmlicher Kommunikationsmechanismen erforderlich wären. Zunächst müsste die zu versendende Objektstruktur in eine für den Transport geeignete Codierung überführt werden. In Anlehnung an die Terminologie des hierfür vorgesehenen Standard Mechanismus, wurde dieser Schritt im Bild mit „Serialisieren“ bezeichnet. Anschließend müssen die zur Codierung verwendeten Daten zur Ziel VM übertragen werden. Nach der Übertragung muss die Objektstruktur ausgehend von der Transportcodierung rekonstruiert werden. Durch die Einfüh-

34. Es gibt einige wenige Ausnahmen, bei denen die Kommunikation über Links von der reinen „Kopiersemantik“ abweicht. Insbesondere können Objekte der Klassen Isolate, Link über Links versandt werden. Darüber hinaus gibt es auch in den Standard Class Libraries noch einige wenige Klassen, deren Objekte eine ähnliche Sonderbehandlung erfahren. Dies betrifft insbesondere Klassen, deren Objekte „Kommunikationsendpunkte“ wie Pipes, Sockets oder Server-Sockets repräsentieren.

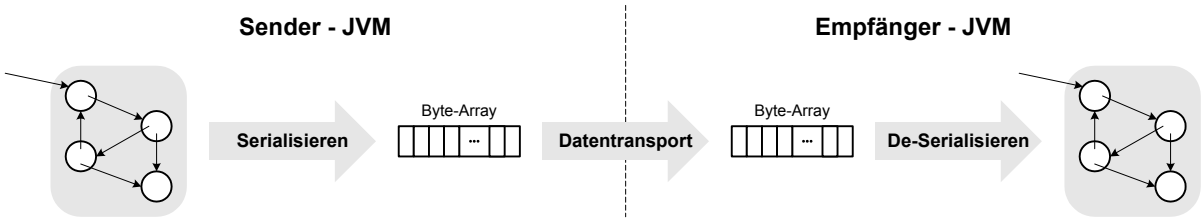


Bild 57: Replikation einer Objektstruktur

rung von Links besteht für den Hersteller der „Multi-Isolate-Java-VM“ die Möglichkeit, den Datentransport zwischen verschiedenen Isolates auf effizientere Weise zu realisieren als dies sonst möglich wäre. Sind die verschiedenen Isolates beispielsweise durch unterschiedliche Prozesse auf einem Rechner realisiert, könnte der Datenaustausch über Links auf Basis von „shared memory“ realisiert werden. Wenn die verschiedenen Java VM’s, die der Realisierung der Isolates dienen, wie im Fall der in Abschnitt 4.2.1.1 vorgestellten Multitasking Virtual Machine, auf Basis eines einzigen Betriebssystemprozesses realisiert sind, entfällt sogar die Notwendigkeit, die Daten zum Zwecke des Transports in eine andere Codierung zu überführen.
Nachdem die Begriffswelt vorgestellt wurde, die dem Java Application Isolation API zugrunde liegt, soll nun noch kurz auf die durch das API definierten Schnittstellen eingegangen werden. Insgesamt besteht die Definition des API aus 16 Klassen[35], die Bestandteil des „java.lang.isolate“ Package sind. Der über diese Klassen zugängliche Funktionsumfang beschränkt sich im Wesentlichen auf drei Dinge:

- Erzeugung und Zerstörung von Isolates sowie die Überwachung des „Lebenszyklus“ von Isolates
- Erzeugung und Zerstörung von Links
- Versand und Empfang von Nachrichten über Links

Spezielle Schnittstellen zur Verwaltung der von den verschiedenen Isolates benutzten Betriebsmittel beinhaltet das Java Application Isolation API nicht.

Weiterer Kapitelaufbau

Wie bereits erwähnt, gab es in der Vergangenheit vielfältige Bemühungen zur Erweiterung der Java VM um ein Prozess Konzept. Der „Java Application Isolation API“ Standard ist nur ein Beispiel hierfür. Die verschiedenen Ansätze unterscheiden sich dabei durch die Art der Realisierung und durch die bereitgestellten Mechanismen zur „Interprozesskommunikation“. Im Zuge der Standardisierung des „Java Application Isolation API“ wurde das Konzept der Links eingeführt, um eine effiziente Kommunikation zwischen verschiedenen Isolates zu ermöglichen. Der Austausch von Nachrichten über Links hat dabei „Kopiersemantik“. Nun hat sich in der Vergangenheit gezeigt, dass im Fall von „safe languages“ effizientere Mechanismen zur Kommunikation zwischen verschiedenen „Prozessen“ realisiert werden können, die eine klare Abschottung der verschiedenen „Prozesse“ gewährleisten, obwohl die Daten der unterschiedlichen Prozesse nicht in unterschiedlichen Adressräumen untergebracht sind. Solche Mechanismen nutzen die Eigenschaft aus, dass „High Level Language“ VM’s zum Zwecke der Wahrung der „type-safety“ bei der Ausführung der „High Level Language“ Programme ohnehin umfang-

35. Die Namen der Klassen lauten: ClosedLinkExeption, Isolate, IsolateEvent, IsolatenEvent.ExitReason, IsolateEvent.Type, IsolateMessage, IsolateMessageDispatcher, IsolateMessageDispatcher.Listener, IsolateMessageVisitor, IsolatePermission, IsolateResourceError, IsolateStartupException, Link, LinkChannel, LinkSerializationException, TransientPrefences

reiche Prüfungen durchführen. Das „Sharing von Objektdaten“ zwischen verschiedenen Prozessen wirft jedoch einige konzeptionelle Probleme auf, die in dem nun folgenden Abschnitt 4.2.2.1 kurz besprochen werden. In Abschnitt 4.2.2.2 werden mehrere in der Literatur beschriebene Systeme vorgestellt, die mit der Entwicklung von Prozess Konzepten für Java in Zusammenhang stehen. Die Darstellung dieser Systeme soll das Spektrum der Möglichkeiten zur Realisierung von Prozess Konzepten für Java aufzeigen.

4.2.2.1 Interprozesskommunikation auf Basis von „shared objects“

„type-safety“ als Ersatz für Adressräume

Klassische Betriebssysteme nutzen das Konzept virtueller Adressräume, um die Daten eines Prozesses gegen unkontrollierte Zugriffe durch andere Prozesse zu schützen. Wie bereits in Abschnitt 2.1.3 dargelegt, ist die Spezifikation der Java VM so gestaltet, dass die Ausführung von Java Bytecode grundsätzlich „memory safe“ ist. Daher kann bei der Realisierung von „Multi-Prozess“ Java VMs darauf verzichtet werden, die Java Heap Daten unterschiedlicher Prozesse in unterschiedlichen Adressräumen unterzubringen, ohne dadurch die wechselseitige Abschottung der Java Prozesse aufzuweichen.[36]

„shared objects“ als Ersatz für „shared memory“

Eine klassische Form der Interprozesskommunikation ist „shared memory“. In Analogie hierzu stellen viele „Multi-Prozess“ Java VMs Interprozesskommunikationsmechanismen bereit, die auf der Nutzung von „shared objects“ beruhen. Das Konzept der „shared objects“ ist jedoch nicht ohne weiteres mit der Konzeption der Java VM vereinbar. Dies soll nun anhand von Bild 58 besprochen werden. Das Bild zeigt zwei Java Prozesse, die über einen Speicher für „shared object“ Daten und einen Kanal miteinander verkoppelt sind. Im Innern der Prozesse ist jeweils ein primitives Modell einer Java VM dargestellt.

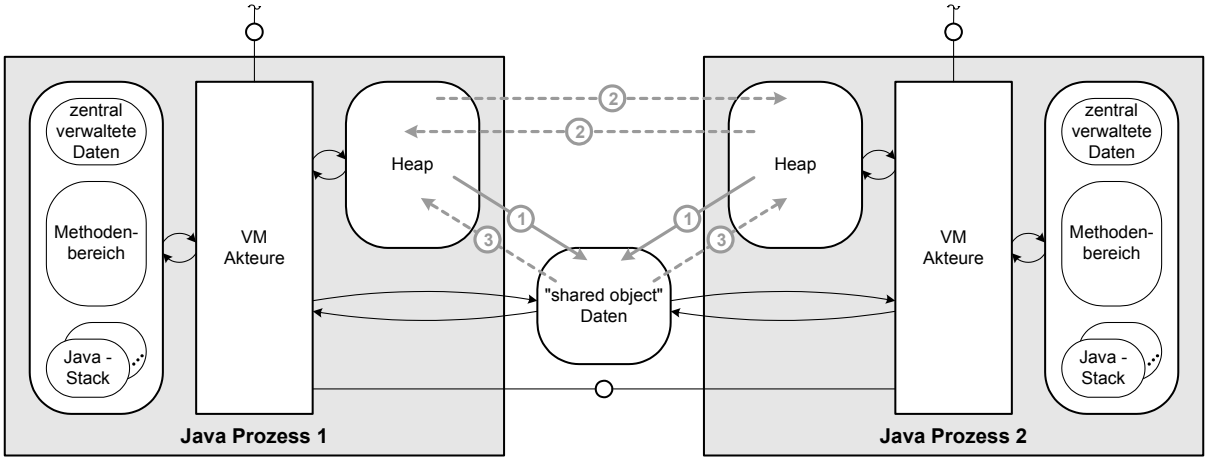


Bild 58: Interprozesskommunikation mittels „shared objects“

Die gesamten im Zugriff der beiden Java Prozesse liegenden Objekte befinden sich zwangsläufig entweder im Zugriff von genau einem der beiden Prozesse oder liegen im Zugriff beider. Daher lag es bei der Darstellung in Bild 58 nahe, die im Zugriff der beiden dargestellten Java

36. Die Eigenschaft der „type safety“ zur Abschottung der Speicherbereiche unterschiedlicher „Systemkomponenten“ einzusetzen ist keine neue Idee. Ein frühes und gleichzeitig sehr bekanntes System dieser Art ist „Pilot“ [Redell et al. 80].

Prozesse liegenden Objektdaten auf drei separate Speicher aufzuteilen: je ein Heap Speicher für jeden Java Prozess und ein Speicher für die Daten der „shared objects“. Obwohl diese Aufteilung für die weitere Diskussion nützlich ist, soll an dieser erwähnt werden, dass sie sich nicht zwangsläufig in den Datenstrukturen einer „Multi-Prozess Java VM“ widerspiegelt:
Aufgrund der Tatsache, dass die Ausführung von Java Bytecode „memory safe“ ist, ist es bei der Implementierung einer „Multi-Prozess Java VM“ nämlich durchaus möglich, nur einen einzigen als Heap organisierten Speicherbereich zur Ablage sämtlicher Objektdaten zu verwenden und die Beziehung von Objekten zu Prozessen dabei außer Acht zu lassen. Diese Vorgehensweise hat sogar gewisse Vorteile und steht nicht im Konflikt mit der wechselseitigen Abschottung der Daten verschiedener Java Prozesse. Ein gravierender Nachteil dieser Form der Implementierung besteht jedoch darin, dass die Ermittlung des Betriebsmittelbedarfs der einzelnen Prozesse erschwert wird. Daher berücksichtigen viele „Multi-Prozess Java VMs“ die Relation zwischen Objekten und Prozessen bei der Verwaltung der Objektdaten und verwalten tatsächlich mehrere Heaps.

Die in der Literatur beschriebenen „Multi-Prozess Java“ Systeme, die das Sharing von Objekten zwischen Prozessen unterstützen, schränken das Sharing von Objekten im Allgemeinen nicht auf Objekte eines fest vorgegebenen Repertoire von Java-Klassen ein, sondern stellen es dem Programmierer frei, eigene Java-Klassen für „shared objects“ zu definieren. Diese Offenheit erfordert es, dass nicht nur die Objekt-Daten der „shared objects“ sondern auch die Typinformation (inklusive des Bytecodes) der „shared objects“ allen beteiligten „Java Prozess VMs“ zugänglich sein muss.

Sharing Modelle

Die in der Literatur beschriebenen „Multi-Prozess Java“ Systeme gehen bei der Realisierung von „shared objects“ auf unterschiedliche Weise vor. Dabei lassen sich im Wesentlichen die beiden in Bild 59 gegenübergestellten Sharing Modelle unterscheiden. In beiden Bildhälften sind jeweils drei Heaps mit Objekten dargestellt. Die Aufteilung der Objekte auf unterschiedliche Heaps soll die Zuordnung des durch die Objekte belegten Speichers zu Prozessen veranschaulichen. Für die weitere Diskussion soll angenommen werden, dass jeder Heap umgekehrt eindeutig einem Prozess zugeordnet ist und dass der durch die darin enthaltenen Objekte belegte Speicher vollständig dem zugehörigen Prozess angerechnet wird.

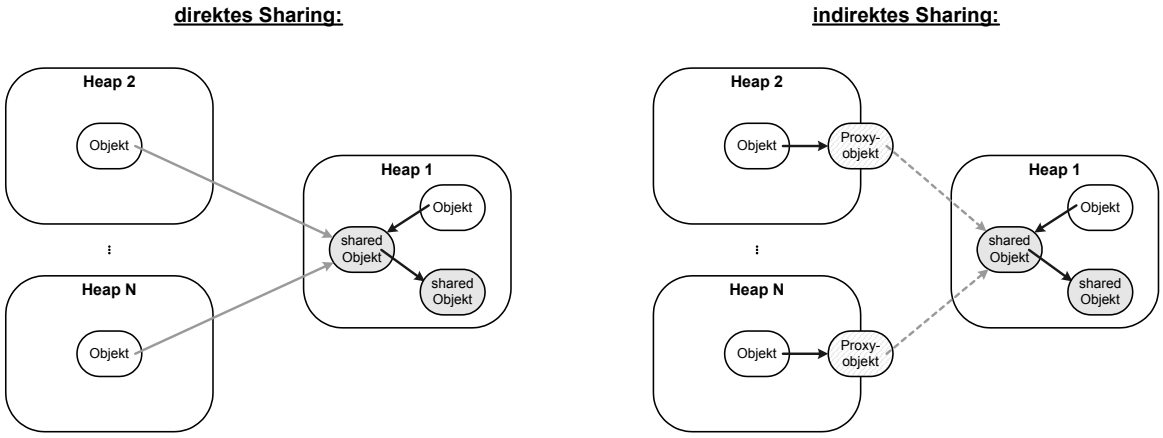


Bild 59: direktes versus indirektes Sharing

Die Idee des „direkten Sharings“ basiert darauf, alle Objekte, unabhängig von ihrer Prozess-Zugehörigkeit, gleich zu behandeln. Mit der Umsetzung dieser Idee geht die Notwendigkeit einher, auch Referenzen zwischen Objekten unterschiedlicher Prozesse genauso zu behandeln wie Re-

ferenzen zwischen Objekten innerhalb eines Heaps. Aufgrund dieser Gleichbehandlung aller Objekte wird dem Modell des direkten Sharings vielfach eine sehr gute Verträglichkeit mit dem Java Programmiermodell bescheinigt.
Die Gleichbehandlung sämtlicher Objekte ist jedoch nicht ohne weiteres mit der Abgrenzung der Betriebsmittel einzelner Prozesse vereinbar. Dies zeigt sich, wenn man der Frage nachgeht, wie bei der Terminierung eines Prozesses mit shared Objekten umgegangen werden soll, die von Objekten anderer Prozesse referenziert werden. Die vielleicht nahe liegendste Verfahrensweise, die betreffenden Objekte einfach zusammen mit dem Prozess zu „vernichten", würde die Implementierung eines „Ausnahme"-Mechanismus erfordern, um die verbliebenen Prozesse, welche noch Referenzen auf die betreffenden shared Objekte besitzen, über die Zerstörung in Kenntnis zu setzen. Abgesehen von der Komplexität die die Implementierung eines solchen „Ausnahme" Mechanismus erfordern würde, steht diese Form der Implementierung zweifelsfrei mit der Idee der Gleichbehandlung aller Objekte in Konflikt. Die in der Literatur beschriebenen Systeme, die das direkte Sharing von Objekten zulassen, umgehen dieses Problem daher durch einen „Trick" und zerstören die shared Objekte nicht explizit, sondern überlassen es einem Garbage Collector diese Objekte zu zerstören, wenn sie nicht länger von einem der übrigen Prozesse referenziert werden. Nun stellt der gerade vorgestellte „Trick" keine wirkliche Lösung des Problems dar, denn er garantiert nicht, dass die betreffenden shared Objekte überhaupt jemals zerstört werden. Darüber hinaus ist das geschilderte Problem nicht ausschließlich auf Objekte beschränkt, die unmittelbar von Objekten anderer Prozesse referenziert werden, sondern betrifft auch alle diejenigen Objekte, die von den direkt referenzierten Objekten erreichbar sind.

Die zweite Form der Realisierung von „shared objects" ist das indirekte Sharing. Systeme, die das Modell des indirekten Sharings implementieren, verbieten die Erzeugung von Heap übergreifenden Referenzen zwischen „gewöhnlichen" Objekten. Stattdessen stellen sie zu diesem Zweck spezielle, vom System verwaltete Proxy-Objekte bereit. Diese Systeme gehen bei der Terminierung von Prozessen typischerweise so vor, dass sie sämtliche in Proxy Objekten gespeicherte Referenzen auf Objekte des zu terminierenden Prozesses als ungültig kennzeichnen, die aktuell in Bearbeitung befindlichen Methodenaufrufe der betroffenen „shared objects" abbrechen und anschließend sämtliche dem Prozess zugeordnete Objekte zerstören. Durch diese Vorgehensweise wird das mit dem direkten Sharing verbundene Terminierungsproblem vermieden. Dies hat jedoch seinen Preis:
Einerseits stellt sich die Nutzung von „shared objects" aus Sicht eines Programmierers wesentlich komplizierter dar. Neben den „shared objects" muss er nun zusätzlich mit Proxy-Objekten umgehen. Desweiteren muss er beim Aufruf der Methoden von „shared objects" stets damit rechnen, dass die Auswertung dieser Methoden vorzeitig abgebrochen wird, weil der Prozess, dem das „shared object" zuzurechnen war, während des Methodenaufrufs terminiert wurde. Andererseits ergeben sich aus diesem Sharing Modell auch Konsequenzen, die die Übergabe von Parametern beim Aufruf der Methoden von „shared objects" betreffen. Es ist beim Aufruf der Methoden von „shared objects" nämlich nicht möglich „Parameterobjekte" „by reference" zu übergeben, da auf diese Art das Prinzip des indirekten Sharings untergraben würde.

Einfluss der automatischen Speicherbereinigung auf die Betriebsmittelverwaltung

Die Speicherverwaltung der Java VM setzt, ebenso wie die Speicherverwaltung anderer High-Level-Language VMs, zwingend einen Garbage Collector zur automatischen Speicherbereinigung voraus. Die automatische Speicherverwaltung hat die Eigenschaft, dass die exakte Bestimmung des aktuell belegten Speichers nur unmittelbar nach einer Garbage Collection bestimmt werden kann. Darüber hinaus verursacht die Durchführung von Garbage Collection Operationen verhältnismäßig großen Aufwand. Im Kontext von „Multi-Prozess Java" Syste-

men stellt sich daher die Frage, wie die Garbage Collection der „Java Process VMs“ organisiert werden soll und woran der aktuelle Speicherbedarf eines Prozesses gemessen werden soll.

Einige Systeme, die das direkte Sharing von Objekten erlauben, gehen bei der Speicherverwaltung so vor, dass sie einen einzigen Heap zur Ablage der Objekte sämtlicher Prozesse verwenden. Durch diese Vorgehensweise werden die Grenzen zwischen den Heaps der einzelnen Prozesse aufgelöst. Dies führt zu der Konsequenz, dass der Heap nur als Ganzes einer Garbage Collection unterzogen werden kann. Ferner ist es bei dieser Form der Verwaltung des Heap Speichers praktisch unmöglich, die zum Zwecke der Garbage Collection aufgewendete Rechenzeit „gerecht“ auf die verschiedenen Prozesse umzulegen. Da der für die Garbage Collection aufgewendete Anteil an Rechenzeit durchaus einen beträchtlichen Teil der gesamten zur Verfügung stehenden Rechenzeit verschlingen kann, steht diese Vorgehensweise grundsätzlich in Konflikt mit der „gerechten“ Aufteilung des Betriebsmittelbedarfs auf Prozesse. Des Weiteren steht sie im Konflikt mit der Implementierung eines prioräts-basierten Shedulings, weil nicht ohne weiteres verhindert werden kann, dass hoch-priore Prozesse zum Zwecke der Garbage Collection angehalten werden müssen, wenn niedriger priorisierte Prozesse große Mengen von Garbage erzeugen.

Viele in der Literatur beschriebene Systeme versuchen diese Probleme dadurch zu vermeiden, dass sie die Gesamtheit der zu verwaltenden Objekte entsprechend ihrer Prozesszugehörigkeit auf unterschiedliche Heaps aufteilen. Sie verwalten üblicherweise je einen Heap pro Prozess und gegebenenfalls weitere Heaps für „shared objects“. Dies eröffnet die Möglichkeit, die Heaps der verschiedenen Prozesse unabhängig voneinander zu bereinigen und den Aufwand an Rechenzeit, der für die Garbage Collection des Heaps eines bestimmten Prozesses aufgewendet wird, unmittelbar dem betreffenden Prozess zuzuordnen. Darüber hinaus stellt die Größe des einem Prozess zugeordneten Heaps ein zweckmäßiges Maß für den durch den betreffenden Prozess belegten Speicher dar. Die Realisierung dieser Form der Speicherverwaltung ist dennoch nicht so einfach, wie es auf den ersten Blick scheinen mag, da Heap-übergreifende Referenzen bei der Garbage Collection berücksichtigt werden müssen. Dabei sind prinzipiell beliebig komplexe Referenzierungsstrukturen zwischen Objekten in unterschiedlichen Heaps denkbar, wenn die Referenzierungsstruktur nicht explizit durch das jeweils implementierte Sharing Modell eingeschränkt wird. Wird die Referenzierungsstruktur zwischen Objekten unterschiedlicher Heaps nicht beschränkt, so erfordert dies den Einsatz entsprechend komplizierter Algorithmen zur „verteilten Garbage Collection“. Hier stellt sich dann erneut das Problem, die für die „verteilte Garbage Collection“ aufgewendete Rechenzeit „gerecht“ auf die vorhandenen Prozesse aufzuteilen.

Prozess Terminierung und „shared objects“

Im Zusammenhang mit der Terminierung von Prozessen stellt sich die interessante Frage, ob ein Prozess auch dann terminiert werden soll, wenn einer der Threads dieses Prozesses gerade mit der Bearbeitung einer Methode eines „shared object“ beschäftigt ist. Eine Möglichkeit besteht darin, die Auswertung der Methode einfach abzubrechen. Nun widerspricht diese Vorgehensweise grundsätzlich der Kapselungsidee, die gerade in „type safe languages“ wie Java besonders stark ausgeprägt ist. Daher finden sich in der Literatur verschiedene Vorschläge zu diesem Thema. Einige Systeme schenken diesem Problem keine Beachtung und brechen die Auswertung solcher Methoden tatsächlich ab. Andere Systeme verschieben die Terminierung eines Prozesses solange, bis die Auswertung sämtlicher Methodenaufrufe von „shared objects“ abgeschlossen ist. Wieder andere schlagen die Einführung eines Transaktionskonzepts vor, um die Daten der „shared objects“ vor Inkonsistenzen zu schützen(siehe [Daynès Czajkowski 02]).

4.2.2.2 Beispiele für „Multi-Prozess“ Java Systeme

In der Literatur lassen sich sehr viele Beispiele für „Multi-Prozess“ Java Systeme finden. Einige dieser Systeme nutzen eine Standard Java als Trägersystem, andere Systeme sind in Form eines Betriebssystemprozesses realisiert und setzen daher ein Betriebssystem als Trägersystem voraus, wieder andere Systeme sind direkt als Java Betriebssysteme konzipiert und stehen in engem Zusammenhang mit der Entwicklung spezieller Mikrokernel. Schon aufgrund der Vielzahl an „Multi-Prozess“ Java Systemen können nicht alle diese Systeme detailliert vorgestellt werden. Daher beschränkt sich die nun folgende Darstellung im Wesentlichen auf eine recht kurze Vorstellung ausgewählter Systeme. Um das Spektrum an Lösungsmöglichkeiten für die im vorangegangenen Abschnitt beschriebenen Probleme bei der Nutzung von „shared objects“ zur Interprozesskommunikation aufzuzeigen, werden jedoch einzelne Aspekte bestimmter Systeme etwas detaillierter vorgestellt.

Java VM basierte „Multi-Prozess“ Java Systeme

Bei „Multi-Prozess“ Java Systemen, die eine „Standard Java“ als Trägersystem nutzen, liegt der besondere Fall vor, dass das Trägersystem bereits in der Lage ist, Java-Bytecode auszuführen. Obwohl dies eine gewisse Erleichterung bei der Implementierung von „Multi-Prozess“ Java Systemen darstellt, kann der Bytecode, der die Programme der einzelnen „Java Process VMs“ darstellt, nicht unverändert von der VM des Trägersystems ausgeführt werden. Vielmehr sind (teilweise sehr aufwendige) Transformationen dieses Bytecodes notwendig, um die Wahrung der Eigenschaften des jeweiligen Prozessmodells sicherzustellen. Da es bei diesen Systemen häufig von großer Bedeutung ist, dass einmal gestartete Prozesse im Verlauf der Programmausführung dynamisch neuen Bytecode nachladen können, werden diese Programmtransformationen üblicherweise erst während des Ladevorgangs des betreffenden Bytecodes durchgeführt. Hierzu nutzen diese Systeme die Möglichkeit, in den Ladeprozess der als Trägersystem verwendeten Java VM einzugreifen[37].

Ein frühes und gleichzeitig sehr bekanntes System dieser Art wurde an der Cornell Universität im Rahmen der beiden Projekte „J-Kernel“ ([Hawblitzel et al. 98], [Hawblitzel 00]) und „JRes“ ([Czajkowski vEicken 98]) entwickelt. Im Rahmen des J-Kernel Projekts wurde die Frage der Abschottung verschiedener Prozesse innerhalb einer Java VM untersucht, während das JRes Projekt sich mit dem Thema „Resource Accounting“ beschäftigt. Der J-Kernel unterstützt das Konzept der „shared-objects“ durch Bereitstellung eines Mechanismus, der das „indirekte Sharing“ von Objekten erlaubt. Bild 60 veranschaulicht die durch den J-Kernel erzwungene, „logische“ Partitionierung des vom Trägersystem bereitgestellten Java Heaps. Die verschiedenen Java-Prozesse werden im Zusammenhang mit dem J-Kernel als Tasks bezeichnet, und die zur Realisierung des indirekten Sharings dienenden Proxy-Objekte werden im Fall des J-Kernels als Capability-Objekte bezeichnet. Das Modell des indirekten Sharings erlaubt es prinzipiell nicht, dass bei Methodenaufrufen von „shared objects“ Parameterobjekte „by reference“ übergeben werden. Um den damit einhergehenden Aufwand für das Kopieren von Parameterobjekten gering zu halten, sieht der J-Kernel eine spezielle Programmtransformation vor. Neben normalen Tasks sind in Bild 60 noch „privileged“ Tasks dargestellt. Die „privileged“ Tasks dienen der Realisierung von zentralen Diensten und stellen den übrigen Tasks Schnittstellen zum Zugriff auf das I/O-System bereit. Da „privileged“ Tasks nicht terminiert werden können und ihre Implementierung außerdem Bestandteil der J-Kernel Implementierung ist, kann die Kom-

37. Die Schnittstelle zum Eingriff in den Lade-Prozess für Java Bytecode wird durch die „Classloader“-Klasse bereitgestellt. Nähere Informationen hierzu sind in: „Dynamic Class Loading in the Java Virtual Machine“ [Liang Bracha 98] zu finden.

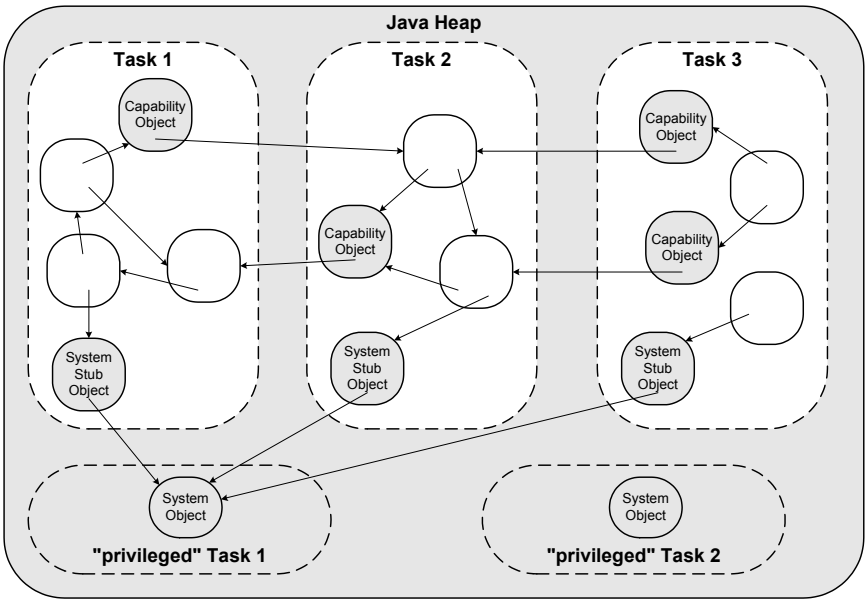


Bild 60: Durch den J-Kernel erzwungene „logische“ Partitionierung des Java Heap

munikation zwischen „normalen“ Tasks und „privileged“ Tasks effizienter implementiert werden. Insbesondere ist es in diesem Fall erlaubt, Parameterobjekte „by reference“ zu übergeben.

Ein weiteres Beispiel für ein „Multi-Prozess“ Java System, welches eine „Standard Java“ als Trägersystem nutzt, ist der „J-SEAL2“ Kernel ([Binder 01], [Binder et al. 01]). Die Entwicklung des J-SEAL2 Kernels geschah in der Absicht, eine Plattform für Mobile Agenten zu schaffen. Ebenso wie der J-Kernel unterstützt der J-SEAL2 Kernel das Konzept der „shared-objects“, durch Bereitstellung eines Mechanismus, der das „indirekte Sharing“ von Objekten erlaubt. Im Gegensatz zum J-Kernel unterstützt er ein hierarchisches Prozess-Konzept. Außerdem ist die Betriebsmittelverwaltung des J-SEAL2 Kernels wesentlich umfassender und ausgereifter als diejenige, die im Rahmen des JRes Projektes umgesetzt wurde.

Bei der Entwicklung des J-Kernels und des J-SEAL2 Kernels wurde dem Fall, dass ein Java Programm mehrfach gestartet wird, keine besondere Aufmerksamkeit gewidmet. Daher sehen diese beiden Kernel kein Code Sharing zwischen unterschiedlichen Prozessen vor. Im Unterschied hierzu wird dieser Fall bei dem in [Czajkowski 00] beschriebenen System explizit berücksichtigt und resultiert in einer völlig anderen Vorgehensweise bei der Programmtransformation.

Als Betriebssystemprozess realisierte „Multi-Prozess“ Java Systeme

In Abschnitt 4.2.1 wurden im Zusammenhang mit der Diskussion von Maßnahmen zur Reduktion des Startupaufwands bereits verschiedene „Multi-Prozess“ Java Systeme vorgestellt, die auf Basis einer oder mehrerer Betriebssystemprozesse realisiert sind.

An dieser Stelle soll noch einmal kurz auf die dort erwähnte Multi-Tasking Virtual Machine (MVM) eingegangen werden. Zu der dortigen Darstellung der MVM muss nämlich noch hinzugefügt werden, dass sie inzwischen stärker in Richtung eines „Multi-Prozess“ Java Systems weiterentwickelt wurde. Insbesondere unterstützt sie inzwischen das in Abschnitt 4.2.2 beschriebene „Java Application Isolation API“. Im Zuge der Integration der Unterstützung für das „Java Application Isolation API“ wurde auch die Frage untersucht, wie die Kommunikation über Links möglichst effizient realisiert werden kann. Näheres hierzu findet man in

[Palacz et al. 02]. Des Weiteren gibt es bereits eine erste Implementierung eines auf das „Java Application Isolation API“ abgestimmten Resource Accounting Frameworks [Czajkowski et al 03].

Bisher wurde noch kein „Multi-Prozess“ Java System vorgestellt, welches das direkte Sharing von Objekten zulässt. Ein solches System ist das an der Universität von Utah entwickelte System mit dem Namen KaffeOS [Back et al. 00]. Die Forschergruppe hat das System mehrfach umbenannt, es war zuvor auch unter den Namen GVM und K0 bekannt. Die Darstellung in Bild 61 zeigt die Heap Struktur von KaffeOS. Das System verwaltet für jeden Prozess einen ei-

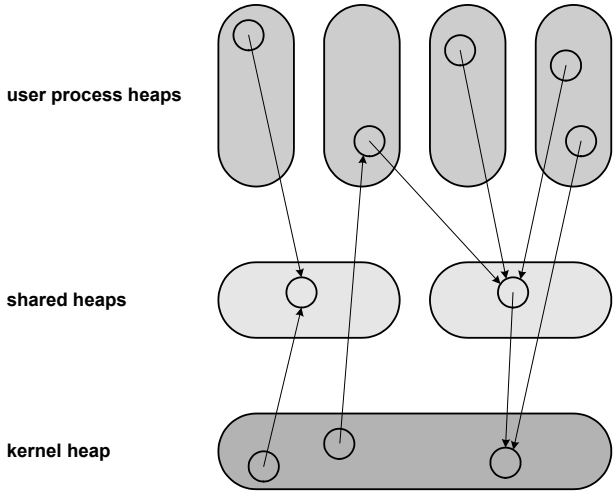


Bild 61: Heap Struktur in KaffeOS (nach [Back et al. 00])

genen Heap und benutzt einen Kernel Heap. Darüber hinaus stellt es einen Mechanismus zur Erzeugung von „shared heaps“ bereit. Um zu erreichen, dass die „user process heaps“ stets unabhängig voneinander bereinigt werden können, lässt das System nur bestimmte Heap-übergreifende Referenzierungsstrukturen zu. Im Einzelnen verbietet das System direkte Referenzen zwischen Objekten verschiedener „user process heaps“ sowie direkte Referenzen, die von „shared heaps“ in „user process heaps“ verweisen. Um die Erzeugung von unerwünschten Referenzen zu verhindern, überwacht das System sämtliche Schreibzugriffe, die den Inhalt der „shared heaps“ betreffen. Wenn der Versuch unternommen wird, eine unzulässige Referenzierungsstruktur herzustellen, so wird dieser Versuch durch eine speziell zu diesem Zweck definierte „Java Exception“ abgebrochen.[38] Die „shared heaps“ enthalten neben den eigentlichen Objektdaten noch weitere Informationen, die für die Nutzung der Heaps durch die „Process VMs“ notwendig sind.[39] Gleich beim Erzeugen eines „shared heap“ wird der betreffende Heap befüllt. Ein einmal erzeugter Heap kann im Nachhinein nicht beliebig verändert werden. Insbesondere ist es nicht möglich, neue Objekte zu diesem Heap hinzuzufügen. Der Datenaustausch erfolgt durch das Lesen und Verändern von Objektattributen sowie durch das Anfordern und Freigeben der Sperren, die den darin enthaltenen Objekten zugeordnet sind.

Java Betriebssysteme

Die Firma Sun war es, die das (vermutlich) erste Java Betriebssystem mit dem Namen „JavaOS“ entwickelte. Dieses Betriebssystem war ein „single user“ Betriebssystem, beinhaltete keinen

38. Da dieses Vorgehen letztlich eine Veränderung der Semantik der Ausführung von Java Bytecode darstellt, ist dieses Vorgehen nicht ganz unproblematisch.
39. Dazu zählen insbesondere die Klassendefinitionen der im Heap vorhandenen Objekttypen.

Compiler und verfügte auch nicht über ein Prozess Konzept. In der Folge implementierte die Firma Sun eine Reihe weiterer Java Betriebssysteme, die Namen wie „JavaOS for Business“ oder „JavaOS for Consumers“ trugen. Inzwischen gibt es eine beachtliche Anzahl leistungsfähiger Java Betriebssysteme, die, wie das „JavaOS for Consumers“, Mikrokernel basiert sind, ausgefeilte Compilertechnik beinhalten und zum Teil auch über leistungsfähige Abschottungskonzepte verfügen. Ein Beispiel hierfür ist das „JX Operating System“ [Golm et al. 02].

Ein weniger neues, aber dennoch recht interessantes Beispiel hierfür ist das an der Universität von Utah entwickelte Betriebssystem „Alta“ [Tullmann 99], welches auf dem ebenfalls dort entwickelten „Flux“ Mikrokernel [Ford et al 96] basiert. Der interessante Aspekt dieses Systems liegt darin, dass der Flux Mikrokernel im Hinblick auf „rekursive Virtualisierbarkeit“ entworfen wurde. Die darauf aufsetzende Implementierung des Java Betriebssystems Alta versucht das Konzept der „rekusiven Virtualisierbarkeit“ in die Java Welt hineinzutragen. Das Ergebnis dieser Überlegungen ist ein hierarchisches Prozesskonzept, welches zugleich als Grundlage für die Betriebsmittelverwaltung dient. Die Java Prozesshierarchie geht bei diesem System mit der in Bild 62 dargestellten Verschachtelung der Heaps der verschiedenen Prozesse einher. Das

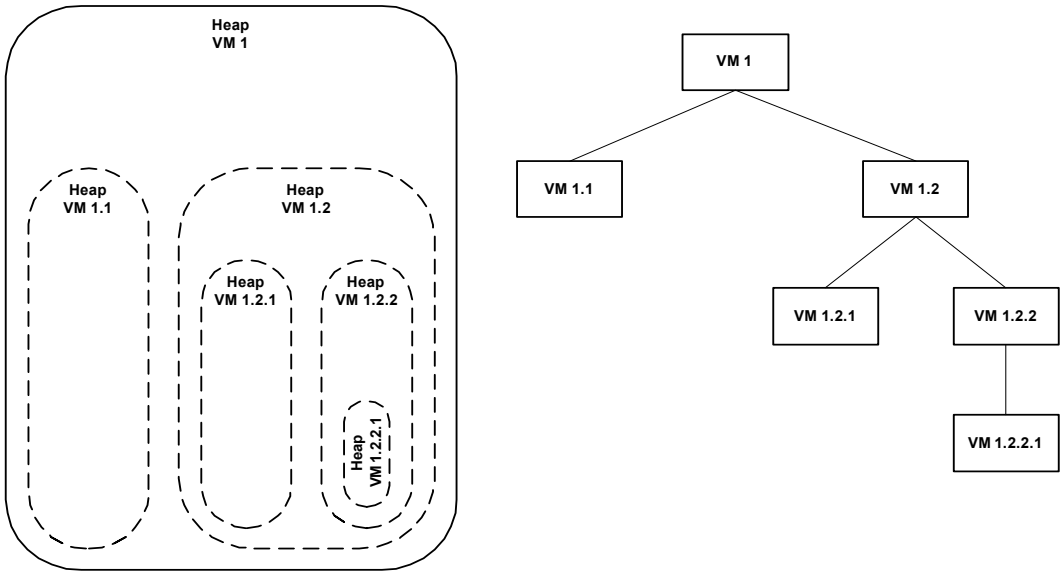


Bild 62: Alta’s Heap Struktur

System stellt einen sehr aufwendigen Mechanismus zur Verfügung, um ein kontrolliertes direktes Sharing von Objekten zwischen beliebigen VM’s zu ermöglichen.

5 Ausblick

Obwohl der Begriff „Virtuelle Maschine“ bereits in den 60er Jahren aufkam, ist die Forschung auf diesem Gebiet keineswegs abgeschlossen, sondern weiterhin ständigen Entwicklungen unterworfen. Speziell die Einführung von objektorientierten Sprachen und der Trend hin zu „safe language environments“ haben in den letzten Jahren wesentlichen Einfluss auf die Konstruktion heutiger Softwaresysteme genommen. Diese Entwicklungen betreffen jedoch nicht nur den Bereich der virtuellen Maschinen, sondern haben viele Querbezüge zum Compilerbau und zur Gestaltung von Betriebssystemen. Aufgrund dieser Vernetzung und der damit einhergehenden gegenseitigen Beeinflussung, unterliegt auch das Thema der Integration virtueller Maschinen einer sehr hohen Evolutionsdynamik. Allein aus diesem Grund besteht in Zukunft weiterer Forschungsbedarf, um die durch die Neuentwicklungen verursachten Konsequenzen für die Integration virtueller Maschinen aufzuzeigen.

Neben dem bereits angesprochenen Bedarf für weitere Arbeiten, der durch zukünftige Entwicklungen begründet ist, besteht auch aufgrund der Komplexität des Themas Anlass für weitere Forschungen. Die vorliegende Arbeit kann nur als erster Schritt zur Auseinandersetzung mit der Thematik angesehen werden. Sie stützt sich vor allem auf das Praxisbeispiel „Integration einer Java VM in den SAP Application Server“, welches durch eine beschränkte Auswahl weiterer Integrationsbeispiele ergänzt wurde, deren Vorstellung sehr knapp gehalten werden musste. Insgesamt konzentriert sich diese Arbeit auf die Darstellung von Beispielen, bei denen es um die Integration einer Java VM geht. Diese weist zwar viele typische Merkmale einer modernen VM auf, ist aber nur eine von vielen praxisrelevanten virtuellen Maschinen. Um die Darstellung der Problematik der Integration virtueller Maschinen abzurunden, ist es erforderlich, weitere Integrationsbeispiele zu betrachten, die andere VMs zum Gegenstand haben. Da die Systeme, die zur Diskussion von Integrationsproblemen betrachtet werden müssen, im Allgemeinen eine sehr hohe Komplexität aufweisen, ist die Diskussion der Integrationsproblematik mit einem hohen Modellierungsaufwand verbunden. Aus diesem Grund sind vermutlich noch viele weitere Arbeiten erforderlich, um die Thematik in angemessenem Umfang darzustellen.

Literaturverzeichnis

[Aho et al. 86] Aho, A. V.; Sethi, R.; Ullman, J. D.;
Compilers: Principles, Techniques, and Tools
Addison-Wesley, 1986

[Alpern et al. 00] Alpern, B.; Attanasio, C.R. ; Barton, J.J.;Burke, M.G.; Cheng, P.; Choi, J.-D.; Cocchi, A.; Fink, S.J.; Grove, D.; Hind, M.; Hummel, S.F.; Lieber, D.; Litvinov, V.; Mergen, M.F.; Ngo, T.; Russell, J.R.; Sarkar, V.; Serrano, M.J.; Shepherd, J.C.; Smith, S.E.; Sreedhar, V.C.; Srinivasan, H.; Whaley, J.;
The Jalapeno Virtual Machine
IBM Systems Journal, vol. 39, no. 1, S. 211-238, Februar 2000

[Alpern et al. 99] Alpern, Bowen; Cocchi, Anthony; Lieber, Derek; Mergen, Mark; Sarkar, Vivek;
Jalapeno - a compiler supported Java virtual machine for servers
ACM SIGPLAN Workshop on Compiler Support for System Software, S. 36-46, Mai 1999

[Arnold 86] Arnold, J.
Shared Libraries on UNIX System V
Summer USENIX Conference, Atlanta, GA, 1986

[Arnold 02] Arnold, Matthew
Online Profiling and Feedback-directed Optimization of Java
Ph.D. thesis, University of New Jersey, Oktober 2002

[Arnold et al. 00-AO] Arnold, M.;Fink, S.; Grove, D.; Hind, M.; Sweeney, P.;
Adaptive Optimization in the Jalapeno JVM
In Proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications, S. 47-65, Oktober 2000

[Arnold et al. 00-S1] Arnold, M.; Hind, Michael; Ryder, Barbara G.;
An Empirical Study of Selective Optimization
In 13th International Workshop on Lanuages and Compilers for Parrallel Computing, August 2000

[Arnold Sweeney 00-S2] Arnold, M.; Sweeney, P. F.;
Approximating the calling context tree via sampling
Technical Report RC 21789, IBM T.J. Watson Research Center, Juli 2000

[Back Hsieh 99] Back, Godmar; Hsieh, Wilson
Drawing the Red Line in Java
Proceeding of the Seventh IEEE Workshop on Hot Topics in Operating systems, Rio Rico, AZ, März 1999

[Back et al. 00] Back, G.; Hsieh, W.; Lepreau, J.
Proceesses in KaffeOS: Isolation Resource Management, and Sharing in Java
4th OSDI, San Diego, CA, 2000

[Bacon et al. 02] Bacon, David F.; Fink, StephenJ.; Grove David
Space- and Time-Efficient Implementation of the Java Object Model
16. Europen Conf. On Object-OrientedProgramming, Juni 2002, LNCS vol. 2374

[Balfanz Gong 97] Balfanz, D.; Gong, L.;
Experience with Secure Multi-Processing in Java
Technical Report 560-97, Department of Computer Science, Princeton University, September 1997

[Bershad et al. 95-PSI] Berhad, B; Savage, S.; Pardyak, P.;Becker, D.; Fiuczynski, M.; Sirer, E.
Protection is a Software Issue
In Proceedings of the Fifth Workshop on Hot Topics in Operating Systems, S. 62-65, Mai 1995

[Bershad et al. 95-SPIN] Berhad, B; Savage, S.; Pardyak, P.; Sirer, E. G.; Fiuczynski, M.; Becker, D.; Chambers, G.; Eggers, S.;
Extensibility, Safety and Performance in the SPIN Operating System
ACM SIGOPS 1995, S. 267 - 284, CO, USA, 1995

[Biliris et al. 94] Biliris, A.; Dar, S.; Gehani, N:; Jagadish, H.; Ramamritham, K.
ASSET: A System Supporting Extended Transactions
ACM SIGMOD Mineapolis, MN, 1994

[Binder 01] Binder, Walter
Designing and Implementing a Secure, Portable, and Efficient Mobile Agent Kernel: The J-SEAL2 Approach
Dissertation, Technische Universität Wien, April 2001

[Binder et al. 01] Binder, W.; Hulaas, J.; Villazon, A.
Portable Resource Control in Java: The J-SEAL2 Approach
16th ACM OOPSLA, Tampa, FL, Oktober 2001

[Borman et al. 01] Borman, S.; Paice, S.; Webster, M.; Trotter, M.; McGuire, R.; Stevens, A.; Hutchinson, B.; Berry, R.
A Serially Reusable Java Virtual Machine for High Volume, Highly Reliable, Transaction Processing
IBM TR 29.3406, Hursley, UK, 2001

[Bryce Razafimahefa 00] Bryce, C.; Razafimahefa C.
An Approach to Safe Object Sharing
15. ACM OOPSLA, Mineapolis, MN,2000

[Bungert 98] Bungert, Andreas
Beschreibung programmierter Systeme mittels Hierarchien intuitiv verständlicher Modelle
Dissertation, Universität Kaiserslautern
Shaker Verlag, Aachen, 1998

[Burke et al. 99] Burke, M.G.; Choi, J.-D.; Fink, S.; Grove, D.; Hind, M.; Sarkar, V. Serrano, M. J.; Sreedhar, V. C.; Srinivasan, H.; Whaley, J.;
The Jalapeno Dynamic Optimizing Compiler for Java
Proceeding ACM Java Grande Conference, San Franzisco, CA, Juni 1999

[Cheng et al. 98] Cheng, P.; Harper, R.; Lee, P.;
Generational Stack Collection and Profile-Driven Pretenuring
In Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation, S. 162-173, Montreal, Canada, Juni 1998

[Choi et al. 99] Choi, Jong-Deok; Gupta, Manish; Serrano, Mauricio; Sreedhar, Vugranam C.; Midkiff, Sam
Escape Analysis for Java
In Procedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications, S. 1-19, November1999

[Click 95] Click,C.;
Combining Analyses. Combining Optimizations
Ph.D. thesis, Rice University, 1995

[Click Cooper 95] Click, Cliff; Cooper, Keith D.;
Combining Analyses, Combining Optimizations
ACM Transactions on Programming Languages and Systems, Vol. 17, No. 2, März 1995, S. 181-196

[Click Paleczny 95] Click,C.; Paleczny, M.;
A Simple Graph-Based Intermediate Representation
In ACM SIGPLAN Workschop on Intermediate Representations , S. 35-49, Januar 1995

[Chrysantis 90] Chrysantis, P.
ACTA: a framework for specifying and reasoning about transaction structure and behavior
ACM SIGMOD, Atlantic City, NJ, 1990

[Czajkowski 00] Czajkowski, Grzegorz
Application Isolation in the Java(tm) Virtual Machine
ACM OOPSLA 00, Mineapolis, MN, Oktober 2000

[Czajkowski Daynès 01] Czajkowski, Grzegorz; Daynès, Laurent
Multitasking without Compromise: a Virtual Machine Evolution
ACM OOPSLA 01, Tampa Florida, 2001

[Czajkowski vEicken 98] Czajkowski, Grzegorz; von Eicken, T.
JRes: A Ressource Accounting Interface for Java
In Proceedings of the 1998 ACM OOPSLA Conference, Vancouver, BC, Oktober 1998

[Czajkowski et al 01-NCI] Czajkowski, Grzegorz; Daynès, Laurent; Wolczko, Mario
Automated and Portable Native Code Isolation

[Czajkowski et al 02] Czajkowski, Grzegorz; Daynès, Laurent; Nystrom, N.
Code Sharing among Virtual Machines
ECOOP'02, Juni 2002

[Czajkowski et al 03] Czajkowski, G.; Hahn, S.; Skinner, G.; Soper, P.; Bryce, C.
A Resource Management Interface for the Java Platform
Sun Microsystems Laboratories Technical Report, SMLI TR-2003-124, May 2003

[Daynès Czajkowski 02] Daynès, Laurent; Czajkowski, Grzegorz;
Lightweight Flexible Isolation for Language-based Extensible Systems
Proc. 28th VLDB Conference, Hong Kong, China, 2002

[Detlefs Agesen 99] Detlefs ; Agesen ;
Inlining of Virtual Methods
Lecture Notes In Computer Science; Vol. 1628, S. 258-278, Springer-Verlag, 1999

[Dillenberger et al. 00] Dillenberger, D.; Bordawekar, R.; Clark, C.; Durand, D.; Emmes, D.; Gohda,O.; Howard, S.; Oliver, M.; Samuel, F.; St.John, R.W.
Building a Java Virtual Machine for server applications: The Jvm on OS/390
IBM Systems Journal, Vol. 39, Nr. 1, 2000

[ECMA 01] ECMA
Common Language Infrastructure (CLI)
http://msdn.microsoft.com/net/ecma, 2001

[Eeckhout et al. 03] Eeckhout, Lieven; Georges, Andy; Bosschere Koen De
How Java Programs Interact with Virtual Machines at the Microarchitectural Level
ACM OOPSLA 2003, Anaheim, CA

[Engelschall 01] Engelschall, Ralf S.
GNU Portable Threads - Manual
http://www.engelschall.com

[Fitzgerald et al. 99] Fitzgerald, Robert; Knoblock, Todd B.; Ruf, Erik; Steensgaard, Bjarne; Tarditi, David
Marmot: An optimizing compiler for Java
Technical Report MSR-TR-99-33, Microsoft research, Juni 1999

[Flatt 99] Flatt, M.; Findler, R. B.; Krishnamurthi, S.; Felleisen, M.;
Programming Languages as Operating Systems (or, Revenge of the Son of the Lisp Machine)
International Conference on Functional Programming (ICFP), Paris, 1999

[Fleury Reverbel 03] Fleury, M.; Reverbel, F.
The JBOSS Extensible Server
Proceedings of Middleware 2003, LNCS 0558, Springer-Verlag, S. 344-373

[FMC 05] Knöpfel, Andreas; Gröne, Bernhard, Tabeling, Peter
Fundamental modeling concepts: Effective Communication of IT Systems
John Wiley & Sons Ltd, England, 2005, ISBN-13: 978-0-470-02710-3

[FMC WWW] FMC Website
Fundamental Modeling Concepts
Hasso-Plattner-Institut für Softwaresystemtechnik, 2003,
URL: http://www.f-m-c.org

[Ford et al 96] Forb, B.; Hibler, M.; Lepreau, J.; Tullmann, P.; Back, G.; Clawson, S.
Microkernels meet recursive virtual machines.
In Proc. of the Second OSDI, S. 137-151, Seattle, WA, October 1996

[Franz 98] Franz, Michael
The Java Virtual Machine: A Passing Fad?
IEEE Software November/Dezember 1998, S. 26-29, 1998

[Franz Kistler 97] Franz, M.; Kistler, T.
Slim Binaries
Communications of the ACM , Vol. 40, Nr. 12, S. 87-94, 1997

[Fraser et al. 92] Fraser, Cristopher W.; Hanson, David R.; Proebsting, Todd A.
Engineering a simple, efficient code-generator generator
ACM LOPLAS 1(3), September 1992

[Gingell et al. 87] Gingell, R.; Lee, M.; Dang, X.; Weeks, M.;
Shared Libraries in SunOS
Summer USENIX Conference, Phoenix, AZ, 1987

[Goldberg 74] Goldberg, Robert P.
Survey of Virtual Machine Research
In: IEEE Computer, S. 34-45, Juni 1974

[Golm et al. 02] Golm, M.; Felser, M.; Wawersich, C.; Kleinoder, J.
The JX Operating System
The USENIX Annual Technical Conference, Monterey, CA, Juni 2002

[Gong 99] Gong, Li
Inside Java 2 Platform Security
Addison Wesley, 1999

[Gosling et al. 96] Gosling, James; Joy, Bill; Steele, Guy
The Java Language Specification
Addison Wesley, 1996

[Gough 01] Gough, John
Compiling for the .NET Common Language Runtime
Prentice-Hall, Upper Saddle River, NJ, 2001

[Gröne et al 02] Gröne, Bernhard; Knöpfel, Andreas; Kugel, Rudolf
Design recovery of Apache 1.3 - A Case study
In: The 2002 International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas NV, USA, S. 87-93, CSREA Press, 2002

[Gröne et al 03] Gröne, Bernhard; Knöpfel, Andreas; Kugel, Rudolf
The Apache modeling project
Hasso-Plattner-Institut für Softwaresystemtechnik, 2003,
URL: http://apache.hpi.uni-potsdam.de

[Hawblitzel et al. 98] Hawblitzel, c.; Chang, C-C.; Czajkowski, G.; Hu, D.; von Eicken, T.
Implementing Multiple Protection Domains in Java
USENIX Annual Technical Conference, S. 259-270, New Orleans, LA, Juni 1998

[Hawblitzel 00] Hawblitzel, Christopher Kirk
Adding Operating System Structure to Language-Based Protection
Ph.D. thesis, Cornell University, August 2000

[Hoare 74] Hoare, C. A. R.
Monitors: An Operating System Structuring Concept
In: CACM 17, Nr. 10, S.549-557, Oktober 1974

[Hecker 81] Hecker, Roland
Herstellungsbeziehungen zwischen Maschinen in Rechensystemen
Dissertation, Universität Kaiserslautern, 1981

[Hennessy Patterson 90] Hennessy, John L.; Patterson, David A.;
Computer Architecture: A Quantitative Approcach
Morgan Kaufman, San Mateo, 1990

[Jordan et al 04-J2EE] Jordan, Mick; Czajkowski, Grzegorz; Daynès, Laurent; Jarzab, Marcin; Bryce, Ciaran
Scaling J2EE Application servers with the Multi-Tasking Virtual Machine
Technical Report, Sun Microsystems, TR-2004-135, 2004
URL: http://research.sun.com/techrep/2004/smli_tr-2004-135.pdf

[JSR-121-03] Java Community Process.
JSR-121: Application Isolation API Specification
http://jcp.org/jsr/detail/121.jsp
(Close of Public Review: 18. März 2003)

[Keller et al 02] Keller, Frank; Tabeling, Peter; Apfelbacher, Rémy et al.
Improving Knowledge Transfer at the Architectural Level: Concepts and Notations
In: The 2002 International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas NV, USA, CSREA Press, S. 101-107, 2002

[Keller, Wendt 03] Keller, Frank, Wendt, Siegfried
FMC: An Approach Towards Architecture-Centric System Development
In: The 10th annual IEEE International Conference and Workshops on the Engineering of Computerbased Systems (ECBS'03), Huntsville AL, USA, IEEE Society, S. 173-182, 2003

[Kennedy Syme 01] Kennedy, A.; Syme, D.;
Design and Implementation of generics for the .NET Common Language Runtime
Proceeding PLDI'01, 2001

[Kistler Franz 97] Kistler, T.; Franz, M.;
A Tree-Based Alternative to Java Byte Codes
Proceedings of the International Workshop on Securety and Efficiency Aspects of Java, Eilat, Israel, 1997

[Kleis 99] Kleis, Wolfram
Konzepte zur verständlichen Beschreibung objektorientierter Frameworks
Dissertation, Universität Kaiserslautern, Shaker Verlag, Aachen, 1999

[Kuck et al 02] Kuck, N.; Kuck, H.;Lott, E.; Rohland, E.; Schmidt, O.
SAP VM Container: Using Process Attachable Virtual Machines to Provide Isolation and Scalability for Large Servers.
Work-in-Progress Session, Java Virtual Machine Research and Technology Symposium, San Franzisco, August 2002

[Liang Bracha 98] Liang, Sheng; Bracha Gilad
Dynamic Class Loading in the Java Virtual Machine
In: ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98), Oktober 1998

[Liang 99] Liang, Scheng
The Java Native Interface
Programmer's Guide and Specification
Addison Wesley, Juni 1999

[Lindholm Yellin 96] Lindholm, T.; Yellin, F.
The Java Virtual Machine Specification
1. Auflage Addison-Wesley, 1996

[Nelson 79] Nelson, P. A.;
A Comparison of PASCAL Intermediate Languages
ACM SIGPLAN Notices 14(8): S. 208-213, 1979

[Oracle 99] **Delivering the Promise of Internet Computing: Integrating Java With Oracle8i**
http://www.oracle.com/database/documents/delivering_the_promise_twp.pdf, April 1999

[Palacz et al. 02] Palacz, K.; Czajkowski, G.; Daynès, L.; Vitek, J.
Incommunicado: Efficient Communication for Isolates
ACM OOPSLA'02, Seattle, WA, November 2002

[Paleczny et al 01] Paleczny, Michael; Vick, Christopher; Click, Cliff;
The Java HotSpot Server Compiler
In USENIX Java Virtual Machine Research and Technology Symposium, April 2001

[Popek Goldberg 74] Popek, Gerald J.; Goldberg, Robert P.
Formal Requirements for Virtualizable Third Generation Architectures
In: CACM, S. 412-421, Juli 1974

[Pugh 99] Pugh, William
Fixing the Java Memory Model
ACM Java Grande Conference, S. 89-98, Palo Alto, CA,Juni 1999

[Ramsey 96] Ramsey, Norman
Relocating Machine Instructions by Currying
ACM SIGPLAN '96, S. 226- 236, 1999

[FRC 2616-HTTP/1.1] The Internet Society
Hypertext Transfer Protocol - HTTP/1.1
http://www.ietf.org/rfc/rfc2616.txt, Februar 1997

[RFC 2109-Cookies] The Internet Society
HTTP State Management Mechanism
http://www.ietf.org/rfc/rfc2109.txt, Februar 1997

[Redell et al. 80] Redell, D. D.; Dalal, Y. K.; Horsley, T. R.; Lauer, H. C.; Lynch, W. C.; McJones, P. R.; Murray, H. G.; Purcel, S. C.;
Pilot: An operating system for a personal computer
In: CACM, 23(2), S. 81-92, Feb. 1980

[Richter 02] Richter, Jeffrey
Applied Microsoft .NET Framework Programming
1. Auflage, Microsoft Press, Redmont Washington, 2002

[Rosenblum et al. 95] Rosenblum, Mendel; Herrod, Stehen A.; Witchel, Emmett; Gupta Anoop
Complete Computer System Simulation: the SimOS Approach
IEEE Parallel and Distributed Technology, Herbst 1995

[Rosenblum et al. 95] Rosenblum, Mendel; Bugnion, Edouard; Devine, Scott; Herrod, Stehen A.
Using the SimOS Machine Simulator to Study Complex Computer Systems
In: ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 1, Januar 1997, S. 78-103

[Rudys et al. 01] Rudys, A.; Clements, J.; Wallach, D.
Termination in Language-based Systems
Network and Distributed Systems Security Symposium, San Diego, CA, Februar 2001

[SAP 90-01] SAP AG,
Berichte der SAP Basis Modeling Group
SAP-AG, Walldorf, 1990-2001.

[Saulpaugh et al. 99] Saulpaugh, Tom; Clements, Tom; Mirho, Charles A.
Inside the JavaOS(TM) Operating System
Addison-Wesley, 1. Auflage, 28. Januar 1999

[Scholz 00] Scholz, Christian
Strategische Organisation: Multiperspektivität und Virtualität
2. Aufl., Moderne Industrie, Landsberg/Lech, 2000

[Seltzer et al. 96] Setzer, M.; Endo, Y.; Small, C.; Smith, K.;
Dealing with Disaster: Surviving Misbehaved Kernel Extensions
In Proc. of the Second Symposium on Operating Systems Design and Implementation, S. 213-227, Seattle, WA, USENIX Association, 1996

[Serrano et al. 00] Serrano, M.; Bordawekar, R.; Midkiff, S.; Gupta, M.;
Quicksilver: A Quasi-Static Compiler for Java
ACM OOPSLA, Mineapolis, MN, 2000

[Singh et al 02] Singh, Inderjeet, Stearns, Beth; Johnson, Mark; and the Enterprise Team
Designing Enterprise Applications with the J2EE™ Plattform, Second Edition
Addison-Wesley, März 2002

[Small Seltzer 96] Small, Christopher; Seltzer, Margo
A Comparison of OS Extension Technologies
Proc. of the 1996 USENIX Technical Conference, S. 41-54, Usenix Assoc., Berkeley, Calif., 1996

[Smith Nair 04] Smith, J. E.; Nair, Ravi
Excerpt from „Virtual Machines: Architectures, Implementations and Applications," to be published by Morgan Kaufmann Publishers , 2004
URL: http://www.ece.wisc.edu/~jes/902/papers/intro.pdf

[Suganuma et al. 00] Suganuma, T.; Ogasawara, T.; Takeuchi, M.; Yasue, T.; Kawahito, M.; Ishizaki, K.; Komatsu, H.; Nakatani, T.;
Overview of the IBM Just-in-Time Compiler
IBM Systems Journal 39, Nr. 1, S. 175-193, 2000

[Suganuma et al. 01] Suganuma, T.; Yasue, T.; Kawahito, M.; Komatsu, H.; Nakatani, T.;
A Dynamic Optimization Franework for a Java Just-In-Time Compiler
ACM OOPSLA '01, Tampa, FL, 2001

[Sun-HSVM] Sun Microsystems
Java HotSpot™ Technology
http://java.sun.com/products/hotspot

[Sun-CDC 02] Sun Microsystems
Porting Guide - Connected Device Configuration and Foundation Profile, Version 1.0.1 (CDC), May 2002
http://java.sun.com/products/j2me, http://java.sun.com/products/cdc

[Suri et al. 01] Suri, N.; Bradshaw, J.; Breedy, M.; Ford, K.; Groth, P.; Hill, G.; Saavedra, R.;
State Capture and Resource Control for Java: the Design and Implementation of the Arome Virtual Machine
Java Virtual Machine Research and Technology Symposium, Monterey, CA, April 2001

[SQLJ] **Embedded SQL for Java**
http://www.sqlj.org/

[Tabeling 02] Tabeling, Peter
Ein Metamodell zur architekturorientierten Beschreibung komplexer Systeme
In: Modellierung 2002, GI-Lecture Notes in Informatics,Tutzing, Springer Verlag, S. 51-61, 2002

[Tomcat] The Apache Software Foundation
Apache Tomcat
http://jakarta.apache.org/tomcat

[Tullmann 99] Tullmann, P. A.
The Alta Operating System
Master's Thesis, University of Utah, 1999
http://www.cs.utah.edu/flux/papers/tullmann-thesis-abs.html

[ucontext 97] The Open Group
„ucontext" manual page
In Single Unix Specification, Version 2, http://www.opengroup.org/onlinepubs/007908799/xsh/ucontext.h.html, 1997

[Wahbe et al. 93] Wahbe, R.; Lucco, S.; Anderson, T.; Graham, S.;
Efficient Software-Based Fault Isolation
14. ACM Symposium on Operating System Principles, S. 203-216, Asheville, NC, Dezember 1993

[Wendt 79] Wendt, Siegfried
The Programmed Action Module: An Element for System Modelling
Digital Processes 1979

[Wendt 91] Wendt, Siegfried
Nichtphysikalische Grundlagen der Informationstechnik: Interpretierte Formalismen
2. Aufl, Springer Verlag, Heidelberg, 1991

[Whaley 03] Whaley, John
Joeq: A Virtual Machine and Compiler Infrastructure
ACM IVME ’03, S. 58-66, San Diego, CA, 2003

[Wilson 92] Wilson, P. R.;
Uniprocessor Garbage Collection Techniques
Proceedings of the 1992 International Workshop on Memory Management, St. Malo, France, September 1992

[Wirth Gutknecht 92] Wirth, N.; Gutknecht, J.;
Project Oberon
ACM Press, New York, NY, 1992

[Wong et al. 03] Wong, Bernard; Czajkowski, Grzegorz; Daynès, Laurent;
Dynamically Loaded Classes as Shared Libraries: an Approach to Improving Virtual Machine Scalability
IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IEEE Computer Society, Washington, 2003

[Zuck 90] Zuck, Wolfgang
Ein Beitrag zur konsistenten Mitdokumentation von Systementwürfen auf der Basis von Strukturplänen
Dissertation, Universität Kaiserslautern, 1990