Eberhard Karls
Universität
Tübingen

Thomas Bitschnau

Performanceuntersuchungen von Java-Transaktionsanwendungen für den WebSphere Application Server auf unterschiedlichen Systemkonfigurationen

Diplomarbeit

eingereicht an der

Universität Tübingen
Fakultät für Informations- und Kognitionswissenschaften

31.01.2008

Betreut von:

Prof. Dr. Wilhelm G. Spruth - Eberhard-Karls Universität Tübingen
Holger Wunderlich - IBM Deutschland GmbH
Martina Schmidt - IBM Deutschland GmbH

Wilhelm-Schickard-Institut für Informatik

Ich versichere, die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Danksagung

Hiermit möchte ich mich bei allen Personen bedanken, die mich während der Erstellung meiner Diplomarbeit unterstützt haben. An erster Stelle steht hierbei meine Mutter, die mein Studium erst möglich gemacht hat. Für seine stete Förderung und Betreuung möchte ich Herrn Prof. Dr.-Ing. Wilhelm G. Spruth danken.

Besonderer Dank gilt auch meinen Betreuern bei IBM, Frau Martina Schmidt und Herr Holger Wunderlich, für ihre tolle Unterstützung und die schöne Zusammenarbeit. Weiteren Mitarbeitern der Firma IBM möchte ich für ihre hilfreichen Beiträge und Unterstützung in administrativen Fragen danken (in alphabetischer Reihenfolge): Uwe Denneler, Elisabeth Puritscher, Michael Stoerchle und Robert Vaupel.

Nicht zuletzt möchte ich mich bei meinen Kommilitonen und den Mitarbeitern des Lehrstuhls *Technische Informatik* der Universität Tübingen für ihre konstruktiven Hinweise und Unterstützung bedanken: Jörg Behrend, Tamer Ergin, Stefan Lämmermann, Maximillian Mittag, Andreas Nagel und Carsten Soemer.

Zusammenfassung

In der vorliegenden Arbeit wird der Zusammenhang zwischen der Konfiguration der Datenbankanbindung eines Anwendungsservers und der CPU-Auslastung auf den beteiligten Servermaschinen untersucht.

Während der Laufzeitanalyse einer *Java Enterprise Edition*-Anwendung durch die Firma *IBM* wurde eine Abhängigkeit der CPU-Auslastung von der Konfiguration der Datenbankanbindung beobachtet. Um dieses Phänomen näher zu untersuchen, wurde eine Testumgebung aufgebaut. Die Testumgebung besteht aus zwei Systemkonfigurationen:

- **Integrierte Konfiguration:** In der integrierten Konfiguration befinden sich der *WebSphere Application Server V6.1 for z/OS* und die *IBM DB2 for z/OS V8*-Datenbank in derselben z/OS-LPAR. Die Anbindung der Datenbank geschieht über einen JDBC-Treiber Typ 2. Das bedeutet, die Kommunikation zwischen Anwendungsserver und Datenbank findet über native Datenbankbibliotheken direkt innerhalb des Hauptspeichers der LPAR statt.

- **Verteilte Konfiguration:** Die DB2-Datenbank verbleibt auch in dieser Konfiguration in der z/OS-LPAR. Der *IBM WebSphere Application Server Network Deployment V6.1* befindet sich innerhalb einer zweiten LPAR. In dieser LPAR ist ein *SuSE Enterprise Linux Server V9R3* als Betriebssystem installiert. Der Zugriff auf die Datenbank findet über einen JDBC-Treiber Typ 4 statt. Anfragen an die Datenbank werden über ein TCP/IP-Netzwerk gestellt, was eine Serialisierung der Anfrage- und Rückgabeparameter notwendig macht.

In den WebSphere-Servern wurde die *Trade6*-Benchmarkanwendung der Firma *IBM* installiert. Mit dem Lastgenerator *jMeter* wurden Testläufe in beiden Szenarien durchgeführt und der Ressourcenverbrauch mit der IBM-Überwachungssoftware *RMF-PM* aufgezeichnet. In den Testläufen wurde eine ansteigende Zahl der gleichzeitigen Benutzer simuliert, was eine steigende Belastung für die Server bedeutete.

Anhand der so erfassten Daten konnte bestätigt werden, dass ein entfernter Zugriff auf die Datenbank aus einer *Java EE*-Anwendung heraus die CPU-Auslastung um bis zu 20% erhöht.

	Durchschnittliche CPU-Auslastung unter Volllast	
Concurrent User	Konsolidiert (JDBC Type 2)	Verteilt (JDBC Type 4)
50	50%	65%
100	95%	115%
150	140%	160%

CPU-Zeit ist ein Kostenfaktor und eine Einsparung in dieser Größenordnung rechnet sich für den Betreiber des Servers.

Eine konsolidierte Konfiguration ist der verteilten Lösung auch in Betrachtung der Antwortzeit und des Durchsatzes überlegen. Der Austausch von Daten zwischen Anwendungsserver und Datenbank innerhalb desselben Hauptspeichers erhöht die Sicherheit der Anwendung, da die Maschinengrenzen nicht überschritten werden. Die Verwendung von z/OS als Betriebssystem für den Betrieb eines WebSphere Application Servers hat weitere Vorteile. Die Nutzung des *Workload Managers* zur Priortätensteuerung von Requests lässt eine feingranulare Verhaltenssteuerung zu, die unter keinem anderen Betriebssystem möglich ist.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Moderne Anwendungen, auf die über das Internet oder ein Intranet zugegriffen wird, werden zunehmend nach dem *Java Enterprise Edition*-Standard (*Java EE*) entwickelt. Die so implementierten Applikationen sind plattformunabhängig und eine offene Schnittstellenarchitektur verspricht eine einfache Erweiterbarkeit und standardisierte Interoperabilität.

Anwendungsserver stellen eine Laufzeitumgebung für Java EE-Anwendungen bereit und befinden sich in einer 3-Tier-Architektur in der zweiten Schicht. Für die Anbindung der in Schicht drei befindlichen Backends gibt es grundsätzlich zwei Möglichkeiten:

- **Integrierter Betrieb:** Anwendungsserver und Backend befinden sich innerhalb derselben Betriebssysteminstanz und tauschen Daten lokal über den Hauptspeicher aus.
- **Entfernter Betrieb:** Anwendungsserver und Backend befinden sich auf getrennten Maschinen. Der Anwendungsserver greift über ein Netzwerk auf das Backend zu.

1.1 Motivation

Während der Untersuchung des Laufzeitverhaltens einer Java EE-Anwendung durch die Firma IBM zeigte sich, dass der Verbrauch an CPU-Zeit auf den Servermaschinen maßgeblich von der Konfiguration der Datenbankanbindung abhängig war. Ein entfernter Zugriff auf die Datenbank schien die Prozessoren der beteiligten Maschinen höher zu belasten, als ein konsolidierter Betrieb der Datenbank.

Ziel der vorliegenden Arbeit ist es, diese Beobachtung zu verifizieren und genauer zu untersuchen. Ob es diesen Unterschied nachweislich gibt und wenn ja, wie dieser zu quantifizieren ist, ist dabei die zentrale Fragestellung. Darüber hinaus soll betrachtet werden, welche Unterschiede sich noch aus einer unterschiedlichen Systemkonfiguration ergeben. Zur Umsetzung dieses Zieles wird eine Testumgebung entworfen und realisiert. Die Testumgebung setzt sich aus zwei möglichen Konfigurationen zusammen. Für die integrierte Lösung befinden sich Datenbank und Anwendungsserver innerhalb derselben z/OS-Betriebssysteminstanz. Für die verteilte Lösung läuft der Anwendungsserver auf einem Linux-Betriebssystem und die über ein Netzwerk verbundene Datenbank verbleibt auf z/OS.

Innerhalb dieser Umgebung werden realitätsnahe Tests durchgeführt und der Verbrauch an CPU-Zeit aufgezeichnet. Darüber hinaus werden weitere Unterschiede, die sich aus der unterschiedlichen Servertopologie und den verwendeten Betriebssystemen ergeben, dokumentiert.

1.2 Aufbau der Arbeit

In Kapitel 2 wird zuerst auf die historische Entwicklung von Anwendungsservern und die Motivation für ihren Einsatz eingegangen. In Kapitel 3 wird der in dieser Arbeit verwendete Anwen-

dungsserver *Web Application Server V6.1*, kurz *WAS*, der Firma IBM im Detail vorgestellt.

In der Testumgebung wird der WAS auf unterschiedlichen Betriebssystemen installiert und untersucht. Kapitel 4 befasst sich mit den Hauptunterschieden, die sich für den WebSphere-Server aus der Verwendung von z/OS bzw. Linux als Betriebssystem ergeben.

Die entworfene Testumgebung und die verwendete Software werden detailliert in Kapitel 5 erläutert. In Kapitel 6 findet eine Auswertung der erfassten Daten statt. Über die ursprüngliche Fragestellung hinaus werden hier weitere Beobachtungen vorgestellt.

Abschließend wird in Kapitel 7, neben einer Bewertung der Ergebnisse, auch ein Ausblick auf zukünftige, mögliche Untersuchungen gegeben.

2 Historie: Von batch zu online

Zu Beginn des Rechenzeitalters in den 40er und 50er Jahren des letzten Jahrhunderts waren Computer große, monolithisch aufgebaute Geräte, die ganze Räume oder sogar Häuser in Beschlag nahmen. Bis in die 60er Jahre hinein konnten mehrere Benutzer den Computer nur nacheinander in einer Art *Stapelbetrieb*, engl. *batch job*, benutzen. Hatte ein Benutzer sein Programm ablaufen lassen, kam der Nächste an die Reihe. Verwaltet wurde der Programmablauf von einem Administrator, der sukzessive die Programme der Benutzer auf dem Rechner ausführte.

Diese Arbeitsweise ist äußerst ineffizient und unpraktikabel. Zum einen konnte nur ein Benutzer zu einem Zeitpunkt am Computer arbeiten und dieser konnte zu einem Zeitpunkt auch nur ein Programm ablaufen lassen. Wenn dieses Programm möglicherweise noch viele Zugriffe auf Peripheriegeräte in seinem Code hatte, dann lag die CPU-Leistung, während auf die Ergebnisse dieser Anfragen gewartet wurde, brach. Eine solche Vergeudung teurer CPU-Zeit war auf Dauer nicht wirtschaftlich.

Mit der technischen Weiterentwicklung der Hardware (z.B. Erfindung des Transistors) und Software (z.B. Multiuser- und Multitasking-Betriebssysteme) wurde die Effizienz der Computer gesteigert. In den 1960er Jahren wurden im Bereich der Betriebssysteme viele Fortschritte gemacht. Die neuen Betriebssysteme erlaubten es, dass zu einem Zeitpunkt mehrere Benutzer mit einem Rechner arbeiteten und diese konnten auch mehrere Programme gleichzeitig ausführen.

IBM führte 1964 mit der S360-Architektur eine Familie von unterschiedlich schnellen Maschinen ein (die schnellste Variante war um Faktor 200 schneller als die langsamste Variante), die erstmals alle dasselbe Betriebssystem und dieselben Peripheriegeräte unterstützten. Für diese Maschinen wurden neue Betriebssysteme entwickelt. Die in MVT (*Multitasking with a Variable number of Tasks*) eingeführten und in MVS (*Multiple Virtual Storage*) bis hin zum heutigen z/OS verwendeten Konzepte, wie virtueller Speicher, Virtualisierung, Partitionierung, Multi-Processing und Multi-Programming, haben auch heute noch eine tragende Bedeutung für moderne Betriebssysteme ([EOO06]).

Eines hatte sich allerdings nicht geändert. Die Geräte waren immer noch sehr teuer und mussten möglichst effizient genutzt werden. Um dies zu ermöglichen, und auch um der räumlichen Verteilung der Nutzer gerecht zu werden, wurde die Terminalarchitektur eingeführt (vgl. Abbildung 2.1). Hier war die Benutzerschnittstelle nicht mehr nur am Gerät selbst zu finden, sondern über ein Netzwerk wurden Ein- und Ausgabeschnittstellen bereitgestellt, über die man auf den Computer zugreifen und ihn steuern konnte. Die Tastatur nahm Eingaben entgegen und auf dem Bildschirm konnte die Programmausgabe betrachtet werden. Es gab auch noch Peripheriegerät, wie zum Beispiel Drucker oder Diskettenlaufwerke.

Man kann in diesem Fall von einer 2-Tier-Architektur sprechen, d.h. man hat zwei Ebenen, aus denen sich die Computernutzung zusammensetzt. Auf der einen Ebene gibt es die Präsentationsschicht mit den Terminals, auf der anderen die Applikationsschicht mit der Servermaschine. In diesem Fall spricht man auch von einem *Client-Server-System* (vgl. Abbildung 2.2). Spricht man heute von einem Client-Server-System, so ist das meist auf die Softwarearchitektur bezogen. Das

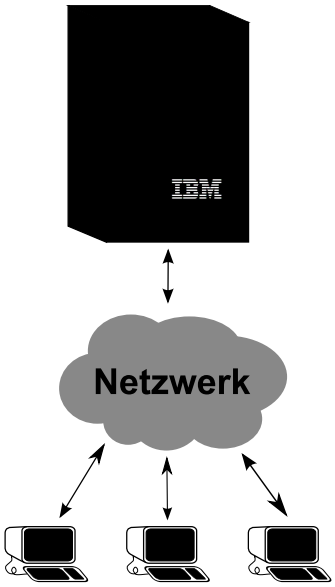


Abbildung 2.1: Terminal-Architektur

Client-Server-System ist ein auch heute noch sehr wichtiges Prinzip und basiert auf der Idee eines Request-Response-Protokolls. Der Client stellt eine Anfrage (Request) an den Server und dieser antwortet (Response) auf diese Anfrage mit dem entsprechenden Ergebnis.

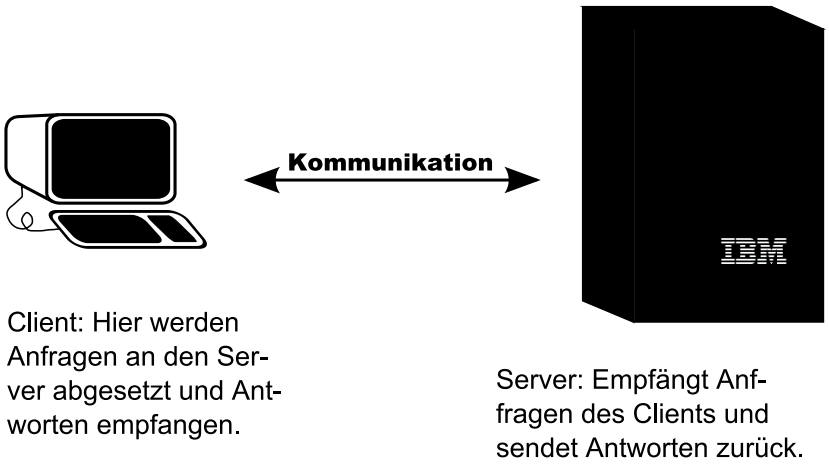


Abbildung 2.2: Client-Server-Architektur

In den 70er Jahren wurden Chips immer preiswerter und konnten in Massenfertigung hergestellt werden. Dies führte zu preiswerteren Computern und der *Personal Computer* (*PC*) fand Einzug in die Gesellschaft. PCs fanden sich im Wohnzimmer, aber auch die Geschäftswelt veränderte sich. Die Clients waren nicht mehr nur für die Darstellung zuständig, sondern übernahmen auch Berechnungen. Das ging so weit, dass teilweise die komplette Berechnung auf den Clients statt fand und der Server nur noch für die Datenhaltung, z.B. in Form einer Datenbank, zuständig war.

In diesem Fall spricht man dann von so genannten *fat clients*. Verarbeiten die Clients nur Ein- oder Ausgabedaten, spricht man von *thin clients*.

Die 2-Tier-Architektur kann man um eine Schicht erweitern. Die Datenhaltung wird dabei in eine extra Schicht ausgelagert und man kommt zu den drei Schichten wie sie in Abbildung 2.3 dargestellt werden.

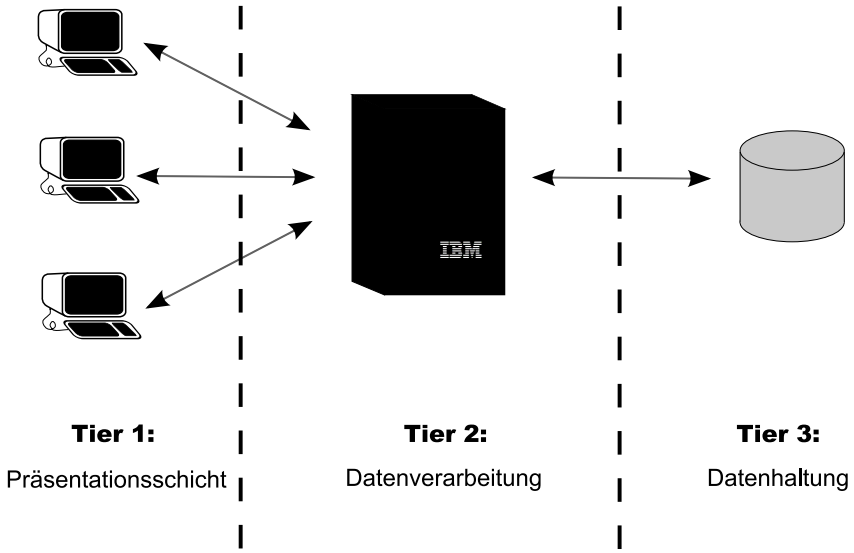


Abbildung 2.3: 3-Tier-Architektur

- ***Präsentationsschicht***: Darstellung von Programmausgaben und Entgegennahme von Benutzereingaben.
- ***Anwendungsschicht***: Auf diesem Tier läuft die eigentliche Applikation und hier werden die Daten aus der letzten Schicht verarbeitet.
- ***Datenhaltungsschicht***: In dieser Schicht findet sich die Software, die die Persistenz der Daten überwacht, z.B. eine Datenbank. Die Server für Persistenz sind nicht zwangsläufig physikalisch von Servern der zweiten Schicht getrennt. Es hat sich aber herausgestellt, dass eine solche Aufteilung des Gesamtvorgangs eine gute Skalierung begünstigt. Die einzelnen Schichten selbst können auf verteilten Systemen laufen um eine weitere Skalierung zu gewährleisten und verteilten Zugriff zu ermöglichen. Mehr dazu in den folgenden Kapiteln.

Damit Client-, Server- und Backendsysteme miteinander kommunizieren können, sind die von einem Betriebssystem zur Verfügung gestellten Funktionen nicht ausreichend. Zu diesem Zweck verwendet man *Middleware*, die schematisch zwischen Betriebssystem und Endbenutzeranwendung angesiedelt ist ([EOO06]). Zur Middleware werden auch noch der Clientanteil, der notwendig ist, um mit dem Server zu kommunizieren, und die verwendeten Kommunikationsprotokolle für den entfernten Aufruf von Methoden, wie z.B. CORBA, RPC, etc. gezählt ([Lan04]). *Application Server* sind eine solche Middleware und es gibt eine Vielzahl von Diensten, die zu Middleware zählen ([EOO06]):

- Datenbanksysteme
- Webserver
- Nachrichtendienste
- Transaktionsmanager
- Java Virtual Machine
- XML-Verarbeitung
- und viele mehr

Genau genommen ist ein Application Server nicht „die“ Middleware, sondern bündelt unterschiedlichste Middleware zu einem großen Paket.

Kommunizieren Client und Server über das Intra- oder Internet und dort verwendete Kommunikationsprotokolle, z.B. das http-Protokoll oder SOAP, spricht man von einem *Web Application Server*. Man kann das zugrunde liegende Modell dann als eine 4-Tier-Architektur beschreiben, in der der Webserver, der http-Anfragen entgegen nimmt, vom Applikationsserver, der entfernte Methodenaufrufe bearbeitet, getrennt wird.

3 (Web) Application Server

Application Server stellen eine Laufzeitumgebung bereit, um serverseitige oder auch verteilte Anwendungen auszuführen.
Dabei sollen Anwendungen folgende Eigenschaften aufweisen ([EOO06]):

- **Funktionalität:** Nutzeranforderungen an die Anwendung müssen erfüllt sein.
- **Zuverlässigkeit:** Gewährleistung der Funktionalität unter sich verändernden Rahmenbedingungen.
- **Useability:** Einfachheit in der Bedienung.
- **Effizienz:** Schonender Umgang mit Systemressourcen.
- **Flexibilität:** Leicht veränder- oder erweiterbar in den Funktionen.
- **Portabilität:** Übertragbar auf unterschiedliche Plattformen.

Ziel ist es, die Anwendungslogik von den darunter liegenden Informationssystemen zu separieren. Man möchte die schematische Aufteilung in Tiers auch programmatisch umsetzen. Das vereinfacht die Entwicklung und Administration solcher Anwendungen und begünstigt eine gute Skalierbarkeit.

Weitere Vorteile einer verteilten Installation einer Anwendung sind nach [Ham05]:

- **Gemeinsame Nutzung von Hardware**: Hardware ist ein großer Kostenfaktor und die gemeinsame Nutzung teurer Hardware spart Kosten.
- **Gemeinsame Nutzung von Daten und Informationen**: Daten beinhalten Informationen, die von Menschen genutzt und bearbeitet werden. Eine zentrale Verwaltung der Daten verringert den Administrationsaufwand und vereinfacht den Erhalt ihrer Konsistenz.
- **Gemeinsame Nutzung von Funktionalität**: Einmal entwickelte Funktionen müssen nicht mehr erneut entwickelt werden, was zu einer Zeit- und Kostenersparnis bei der Entwicklung führt. Desweiteren kann davon ausgegangen werden, dass häufig verwendete Funktionen mit der Zeit fehlerfreier sind, als neu entwickelte.

Dabei arbeitet eine Vielzahl von Modulen zusammengefasst, um eine Anfrage an die Anwendung zu bearbeiten. Diese Module ergeben zusammengefasst einen *Application Server*.

Obwohl der Begriff Application Server plattformunabhängig ist, sind heutzutage meist serverseitig ausgeführte Java-Laufzeitumgebungen gemeint, welche die *Java Platform, Enterprise Edition*-Spezifikation von SUN umsetzen, und in denen die Anwendung abläuft. Insbesondere soll dem Entwickler durch das Komponentenmodell in Java die Arbeit erleichtert werden. Die Java-Bausteine, die dem Komponentenmodell gehorchen, werden *Beans* genannt.

3.1 Java EE - Java Platform, Enterprise Edition

Seit kurzem wird die unter dem Akronym *J2EE* (*Java 2 Enterprise Edition*) bekannte Spezifikation nur noch *Java Platform, Enterprise Edition*, kurz *Java EE*, genannt ([JEE08a]). Die in dieser Arbeit verwendete Software basiert noch auf dem J2EE-Standard 1.4, es wird aber in der Folge der neue Begriff Java EE verwendet.

Java EE basiert auf der *Java Standard Edition*, kurz *Java SE*. Java SE ist eine Sammlung von APIs für Java von SUN. Abbildung 3.1 ([JSE08]) zeigt die Komponenten aus denen sich Java SE zusammensetzt.

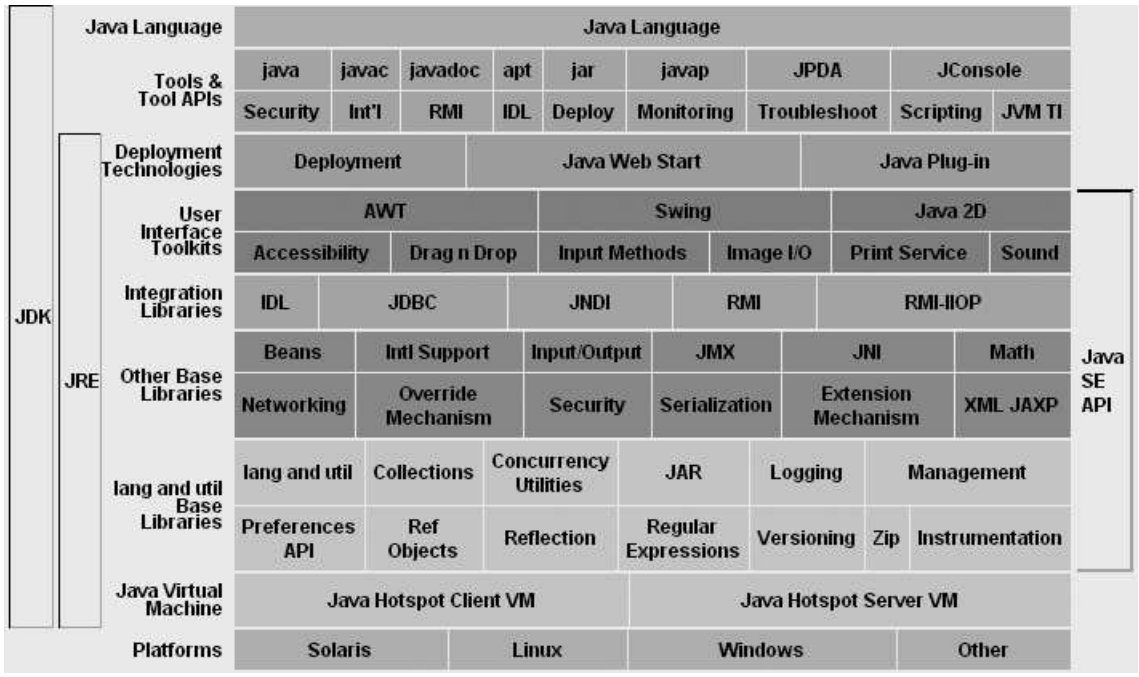


Abbildung 3.1: Java 2 Standard Edition Technologie

Die Implementierung der *Java Virtual Machine*, kurz *JVM*, auf der in Java geschriebene Programme interpretativ ablaufen, wird dabei von den verschiedenen Computerherstellern übernommen. Deren Umsetzung der JVM wird von SUN zertifiziert.

Java EE ist die Definition von Schnittstellen und Standards, die den Java-Programmierer in die Lage versetzen, serverseitige und verteilte Anwendungen zu entwickeln. Java SE ist im Gegensatz dazu darauf ausgerichtet grafische *rich clients* zu entwickeln. Java EE erweitert das bestehende Java SE SDK (*Software Development Kit*) um ([JEE08a]):

- Webservices
- Ein erweitertes Komponentenmodell
- Management-APIs
- Kommunikations-APIs

Abbildung 3.2 fasst zusammen, wie Java EE logisch zu verstehen ist ([Sha03]). Die Abbildung ist kein Indiz für den physikalischen Aufbau der verteilten Anwendung.

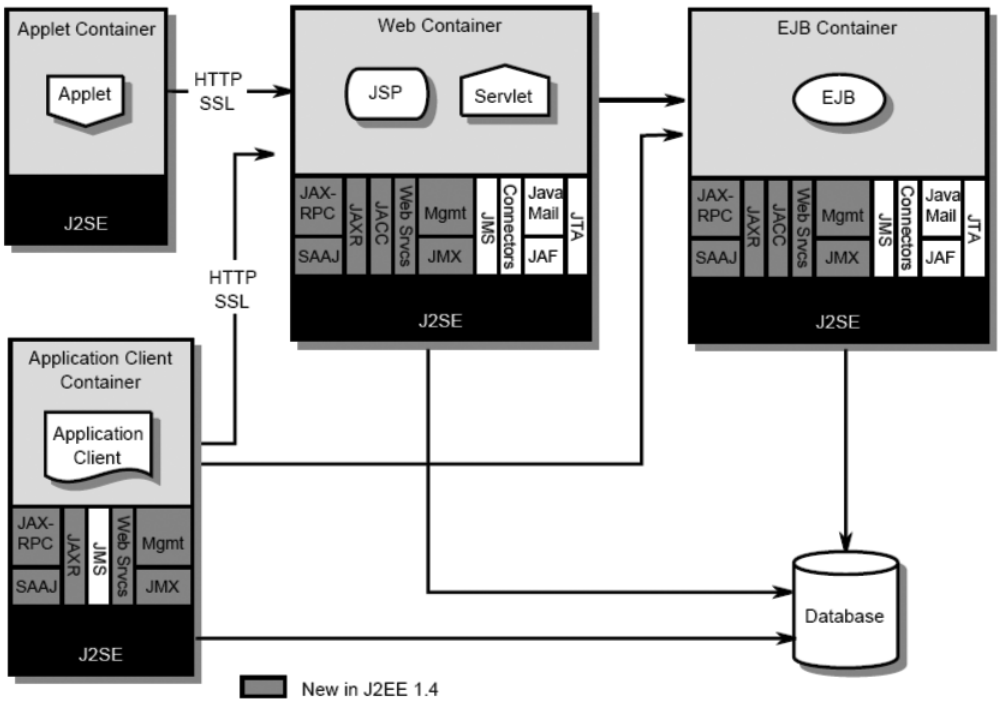


Abbildung 3.2: Die Java-EE-Spezifikation

Verteilte Geschäftsanwendungen sollen sicher, stabil und portabel sein. Mit Java EE möchte SUN diesen Anforderungen gerecht werden. Java EE ist allerdings kein Produkt im Sinne einer Implementierung, sondern eine Spezifikation von Technologien. Die Implementierung wird von anderen Herstellern übernommen, deren Produkte dann von SUN durch Tests mit einem *Java Compatible Enterprise Edition*-Zertifikat verifiziert werden. Da das ganze Modell modular und komponentenbasiert ist, trägt SUN mit seinem Zertifikat der Austauschbarkeit der Komponenten Rechnung. Durch die Definition von standardisierten Schnittstellen, und der Verifizierung dieser, lässt sich die Implementierung einfach erweitern und austauschen. Hier findet sich das Wiederverwendbarkeitsparadigma „Write once, run anywhere“ von SUN wieder ([Sta05]).

Es werden im folgenden die wichtigsten Teile von Java EE kurz vorgestellt. Auf weitergehende Teile der Architektur wird im Verlauf der Arbeit bei Bedarf eingegangen.

Applets

Sie stellen eine besondere Form eines Clients dar. Applets liegen als vorkompilierter *Byte Code* auf dem Server bereit, werden zur Benutzung auf den Client heruntergeladen und auf dem Client ausgeführt. Der Benutzer benötigt zur Interpretation des Byte Codes eine vorinstallierte Java-Laufzeitumgebung.

Java Server Pages

Inhalte, die im Internet zur Verfügung gestellt werden und die ein Benutzer in seinem Browser dargestellt bekommt, werden von *Webservern* geliefert. Über das http-Protokoll stellt der Benutzer eine Anfrage an einen Server und bekommt das Ergebnis in Form einer HTML-Seite vom Server geliefert. Der Browser setzt den HTML-Code in die entsprechende Ansicht um, die der Benutzer dann am Bildschirm präsentiert bekommt. Der hier beschriebene Ablauf war zu Beginn des Internets statisch, d.h. die Webseiten lagen in ihrer Textform in einem Verzeichnis auf dem Server und blieben solange gleich, wie die Datei nicht verändert wurde. Für dynamische Daten, wie z.B. Kontodaten bei einer Bank, ist dieses Vorgehen problematisch.

Ein erster Schritt diesem Problem Abhilfe zu schaffen sind serverseitige Skripte, sogenannte *Server Side Includes*, kurz *SSI*. SSIs können eine Webseite oder Teile davon zum Zeitpunkt des Abrufs dynamisch, unter Zuhilfenahme von Persistenztechnologien erstellen. SSI-Techniken dieser Art gibt es einige. So zählen u.a. CGI, PHP, Active Server Pages von Microsoft oder die hier vorgestellten *Java Server Pages*, kurz *JSP*, dazu.

Hierbei besteht die angeforderte HTML-Seite nicht mehr nur aus reinem HTML-Code. Die angeforderte Seite enthält Anweisungen, die auf dem Server ausgeführt werden. Diese Anweisungen liefern zur Laufzeit ein Ergebnis, das in der HTML-Seite den Platz der Anweisung einnimmt. Im Falle von Java Server Pages sind diese Anweisungen in Java geschrieben, der in skriptähnlicher Form in den HTML-Code eingebettet ist.

```
<HTML>
    <BODY>
        Hello! The time is now <% = new java.util.Date() %>
    </BODY>
</HTML>
```

Listing 3.1: Ein einfaches Beispiel einer JSP ([Spr06])

Auf dem Server wird durch diesen Aufruf in Listing 3.1 ein neues Java-Objekt erstellt, das den Zeitpunkt des Aufrufs repräsentiert. Das in den Tags <% und %> eingeschlossene Java-Kommando entspricht dem Javacode

```
System.out.print (new java.util.Date());
```

JSPs können weitaus komplexer sein und besitzen noch mächtigere Direktiven. So können z.B. auch die eingangs erwähnten Beans aus dem HTML-Code heraus aufgerufen werden, was die folgenden beiden Listings zeigen:

```
package example.j2ee.beans;

import java.util.Date;

/* Verwaltet die derzeitige Uhrzeit */

public class GimmeDate {

    Date date;

    public GimmeDate(){ };

    public String getTime() {
```

```
        date = new Date();
        return date.toString();
    }
}
```

Listing 3.2: Erstellung einer einfachen Bean...

```
<HTML>
    <BODY>
        <!-- Initialisierung des Beans -->
        <jsp:useBean id="time" class="example.j2ee.beans.GimmeDate"/>

        <!-- Aufruf des Beans -->
        Hello! The Time is now
            <jsp:getProperty name="time" property="Time"/>
    </BODY>
</HTML>
```

Listing 3.3: ...und seine Verwendung in einer JSP.

Auf Beans wird zu einem späteren Zeitpunkt noch einmal genauer eingegangen (S. 13).

Serverseitige Applikationen übersteigen die Fähigkeiten eines einfachen Webservers. Der Webserver muss, im Falle von Java, um eine *Servlet-Engine* oder auch einen *Servlet-Container* erweitert werden. Um den Code auszuführen wandelt der Webserver die JSP in eine Java-Klasse um, deren Ausgabe wiederum HTML-Code ist. Diese serverseitigen Java-Klassen nennt man *Servlets*.

Servlets

Historisch betrachtet gab es die Servlets vor den JSPs. Die JSPs werden von einer JSP-Engine in ein Servlet umgesetzt, womit klar ist, dass Servlets dasselbe können wie JSPs (aber nicht umgekehrt!) ([Sta05]).

```
package example.j2ee.servlets

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public class SimpleServlet extends HttpJspBase {

    public void _jspService(HttpServletRequest request,
                            HttpServletResponse response)
                throws java.io.IOException, ServletException {
        /***
        * Der Übersichtlichkeit halber werden hier für das
        * Sessionmanagement, das Abfangen von Exceptions und
        * die Verbindung benötigte Aufrufe  nicht dargestellt
        **/

        try {
            out.write("<HTML>\r\n");
            out.write("<BODY>\r\n");
            out.write("Hello! The Time is now ");
            out.print( new java.util.Date() );
```

```
            out.write("</BODY>\r\n");
            out.write("</HTML>\r\n");
        } catch (Throwable t) {
            //dothisandthat
        }
    }
}
```

Listing 3.4: Das Servlet-Prinzip

Listing 3.4 zeigt wie das Java-Servlet als Ausgabe eine HTML-Seite mit dynamischem Inhalt erzeugt.
JSPs können einfacher sein als Servlets, haben aber nicht den gleichen Funktionsumfang ([Sta05]):

- JSPs werden sehr schnell unübersichtlich, wenn sie komplexe Java-Strukturen enthalten.
- Die Variable `out` (Klasse javax.servlet.jsp.JspWriter) kann keine Binärdaten wie Bilder oder PDF-Dokumente ausgeben.
- JSPs sind sehr stark auf HTML spezialisiert, andere Textformate, wie z.B. CSV, sind mit Servlets leichter zu realisieren.
- Servlets können unterschiedliche HTTP-Anfragen, wie `Post` oder `Get`, behandeln.

Kurz gesagt: Wenn es um Darstellung geht, sind JSPs das Mittel der Wahl, geht es um Funktionalität, bieten sich Servlets an ([Sta05]).

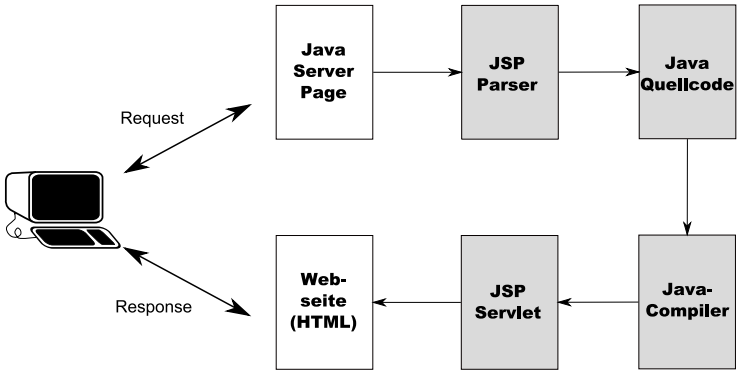


Abbildung 3.3: Der Kreislauf eines JSP-Aufrufs

In der Praxis werden JSPs und Servlets meist gemeinsam verwendet. Der Client ruft eine JSP auf und gibt dabei mögliche Parameter mit. Aus den in die JSP eingebetteten Kommandos werden Servlets erstellt. Diese werden ausgeführt und erzeugen als Ergebnis eine HTML-Seite.

Abbildung 3.3 zeigt den Kreislauf eines solchen Aufrufs ([Spr06]). Die im Bild grau dargestellten Bestandteile sind die eingangs erwähnten Module und bilden einen Web Application Server. Das Modul, welches die Laufzeitumgebung für Servlets und JSPs bereitstellt, wird als *Web Container* bezeichnet.

Web Application Server sind in der Praxis, im Bezug auf die Anzahl der verwendeten Module und Technologien, deutlich komplexer als in Abbildung 3.4 dargestellt. Man möchte meistens

nicht nur das aktuelle Datum ausgeben, sondern zum Beispiel Daten aus verteilten Datenbanken transaktionssicher verarbeiten und dabei noch mit anderen Servern nachrichtenbasiert kommunizieren.

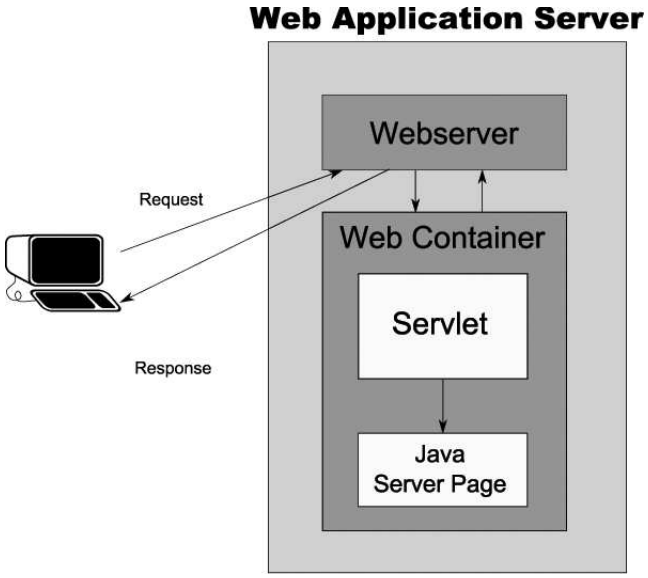


Abbildung 3.4: Web Application Server ([Spr06])

Enterprise JavaBeans

Nach SUN sind JavaBeans wiederverwendbare Softwarebausteine, die über eine grafische Oberfläche bearbeitet werden. Ursprünglich wurden Beans im Zusammenhang mit der Programmierung von grafischen Oberflächen verwendet. Dabei sind sie Komponenten einer grafischen Oberfläche, die sich in ihren Eigenschaften, engl. *Properties*, zum Beispiel Größe oder Inhalt, verändern können. Im Zusammenhang mit Java EE versteht man unter einer JavaBean eine wiederverwendbare Softwarekomponente, die verschiedene Objekte sinnvoll bündelt, durch einen Namen identifizierbar ist und in ihren Eigenschaften verändert werden kann. Eine JavaBean kann z.B. eine Adresse repräsentieren und bündelt dabei Objekte wie `Name`, `Straße`, `Wohnort`, usw. Diese Objekte mit ihren Instanzvariablen sind dann die Eigenschaften der JavaBean. Im Prinzip kann man eine JavaBean also als Datencontainer bezeichnen ([Sta05]).

Enterprise JavaBeans, kurz *EJB*, sind ebenfalls Komponenten. Allerdings werden sie verwendet, um verteilte, transaktionslastige Geschäftsanwendungen zu implementieren, und sind als Teil der Anwendungslogik und nicht nur als Datencontainer zu betrachten. Innerhalb eines Web Application Servers haben sie ihre eigene Laufzeitumgebung, den *EJB-Container*. Über einen entfernten Prozeduraufruf, engl. *Remote Procedure Call*, führen sie den in ihren Methoden implementierten Service aus. *Remote Method Invocation*, kurz *RMI*, ist das für den entfernten Prozeduraufruf zuständige Middlewareprotokoll, wenn Client und Server in Java geschrieben sind. Wenn ein Programmteil nicht in Java implementiert wurde, kann *RMI/IIOP* (gesprochen RMI over IIOP) verwendet werden. Das *Internet Inter Orb Protocol* ist ein Standard der Object Management Group

welches im Rahmen des *CORBA*-Standards (*Common Object Request Broker Architecture*) entwickelt wurde. Kurz gesagt lassen sich mit CORBA unter Zuhilfenahme eines *Object Request Brokers*, kurz *ORB*, Methoden verteilter Anwendungen in heterogenen Umgebungen aufrufen. Der ORB fungiert hierbei als Anlaufstelle für die Methodenaufrufe und übernimmt die Verwaltung der Aufrufe im Kontext des jeweils aufgerufenen Objekts. Dabei verbirgt der ORB Eigenheiten der verwendeten Programmiersprache vor der aufrufenden Gegenstelle und umgekehrt. Das zur Kommunikation verwendete Protokoll ist das oben erwähnte IIOP.

Der Web Application Server aus Abbildung 3.4 kann also auch noch einen EJB-Container enthalten:

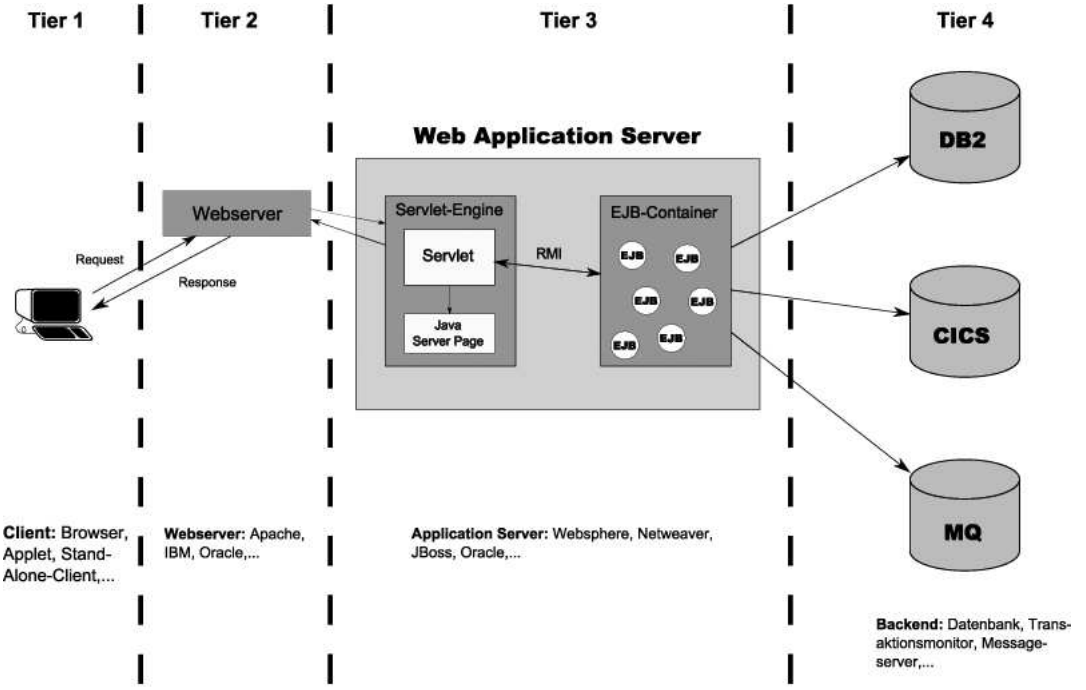


Abbildung 3.5: 4-Tier-Architektur

Abbildung 3.5 zeigt, wie in einer Client-Server-Architektur der Server auch zum Client werden kann. Der Webserver stellt stellvertretend für den „eigentlichen“ Client die Anfragen an den Web Application Server. Für den Benutzer bleibt transparent, woher die ihm letztendlich angezeigte Seite kommt und welche Technologien für ihre Erstellung verwendet wurden. In der Literatur wird daher auch manchmal der Webserver als erstes Tier gezählt ([uFA+06]).

Ein weiterer Aspekt an EJBs ist die Mächtigkeit des Containers. Er nimmt dem Entwickler Arbeit ab und stellt u.a. folgende Services zur Verfügung ([Sta05]):

- **Überwachung des Lebenszyklus von EJBs**: EJBs durchlaufen verschiedene Zustände innerhalb ihrer Existenz. Das dabei nur gewollte Übergänge stattfinden, stellt der Container sicher.
- **Instanzen-Pooling**: Eine EJB kann in mehreren Instanzen existieren, diese verwaltet der Container in einem Pool.

- **Namens- und Verzeichnisdienst**: Die EJBs lassen sich über ihren Bezeichner ansprechen und damit das funktioniert, übernimmt der Container den Namensdienst.
- **Transaktionsdienst**: Die Aufgaben von EJBs bestehen in der Regel aus mehreren Teilschritten. Dabei müssen die *ACID*-Eigenschaften gewährleistet sein.

  ACID ist ein akronym für *Atomizität*, *Konsistenz* (engl. *Consistency*), *Isolation* und *Dauerhaftigkeit*. Diese vier Eigenschaften sind die Grundeigenschaften, die eine korrekt durchgeführte Transaktion erfüllen muss.
- **Nachrichtendienst**: Hiermit können EJBs nicht mehr nur über entfernte Prozeduraufrufe, sondern auch über den Java-eigenen Nachrichtendienst *Java Message Service*, kurz *JMS*, asynchron kommunizieren.
- **Persistenz:** Damit ist zuerst einmal nicht gemeint, dass die Daten, die innerhalb des Beans bestehen oder vom Bean verarbeitet werden, persistent gespeichert werden. Darum kümmert sich ein Bean selbst. Der Container überwacht nur, dass Beans, die durch Paging oder durch einen Containerneustart auf Festplatte ausgelagert werden, in dem Zustand wieder hergestellt werden, in dem sie vor der Auslagerung waren. Manche Beans beherrschen das selbst, man spricht dann von *Bean Managed Persistence*, andere verwaltet dabei der Container (*Container Managed Persistence*).

Man unterscheidet drei Arten von EJBs ([Sta05]):

- **Entity-Beans:** Sie bilden das Datenmodell ab, repräsentieren also ein Objekt der realen Welt, z.B. ein Bankkonto.
- **Session-Beans:** Sie beinhalten Geschäftslogik, z.B. „Überweise Betrag A von Konto B nach Konto C“, und verwenden dabei die Entity Beans.

  Session Beans existieren in zwei Ausprägungen:
  - **Stateless Session-Beans:** Man kann sich die Methoden einer Stateless Session-Bean als statische Objektmethoden vorstellen. Sie sind nicht in der Lage sich einen Client zu merken oder einen Client wieder zu erkennen, sie sind also zustandslos. Ein Beispiel für ein Stateless Session-Bean ist z.B. eine Überweisung. Das Session-Bean, das diese Transaktion durchführt, vergisst nach Beendigung der Transaktion alle verwendeten Parameter wieder und steht dann einer neuen Anfrage zur Verfügung. Sie lassen sich vom EJB-Container bequem in einem Pool verwalten und können so ressourcensparend wiederverwendet werden.
  - **Stateful Session-Beans:** Sie merken sich ihren Zustand in ihren Instanz-Variablen und sind so in der Lage einen Client wieder zu erkennen. Ein Beispiel für die Verwendung eines Stateful Session-Beans ist z.B. ein Warenkorb für eine Online-Bestellung, der über die komplette Sitzungsdauer gespeichert wird. Eine Wiederverwendung des Stateful Session-Beans ist somit nicht möglich, da sich der innere Zustand der Bean von Instanz zu Instanz unterscheidet.
- **Message Driven Beans:** Beans, die asynchrone Kommunikation zwischen den Komponenten via JMS unterstützen. Dabei können zwei Arten von Kommunikation eingesetzt werden. Zum Einen die *Point-to-Point-Kommunikation*, bei der ein Bean gezielt mit einem

anderen Bean Nachrichten austauscht. Zum Anderen die sogenannte *Publish-Subscribe-Kommunikation*, bei der ein Bean eine Nachricht absetzt und alle Beans, die sich zuvor als Empfänger der Nachrichten dieses Beans registriert haben, die Nachricht erhalten.

Die oben beschriebenen Technologien bilden den Kern von Java EE. Sie folgen dem immer gleichen Konzept, welches in Abbildung 3.6 dargestellt ist. Komponenten laufen in Containern ab, welcher Schnittstellen nach außen anbietet, um die Komponenten nach außen verfügbar zu machen. Über Konnektoren können Beans aus dem Container heraus auf die Außenwelt zugreifen.

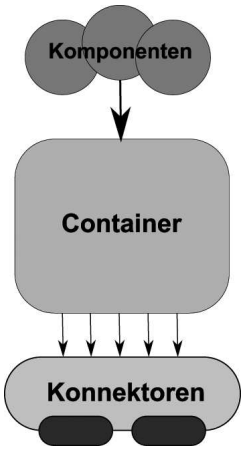


Abbildung 3.6: Java EE-Konzept

Aktuelle Versionen der hier behandelten Java EE-Komponenten sind ([JEE08a]):

- Java Platform, Enterprise Edition 5
- Enterprise JavaBeans 3.0
- JavaServer Pages 2.1
- Java Servlet 2.5

Aktuelle Web Application Server verwenden teilweise noch vorherige Versionen der Spezifikation. Auf mittels Java EE implementierte Anwendungen kann auf verschiedene Arten zugegriffen werden, welche im folgenden kurz vorgestellt werden.

3.1.1 Tier-1-Software

Es gibt grundsätzlich drei Ansätze um Java EE Anwendungen zu nutzen ([JEE08b]):

- **Web Clients:** Sie bestehen aus zwei Teilen. Auf der einen Seite gibt es die dynamischen Webseiten, die auf dem Server liegen und deren Inhalt auf dem Servertier erzeugt wird. Auf der anderen Seite der Browser des Benutzers, der die so erzeugten Seiten darstellt. Außerdem nimmt der Browser Eingaben entgegen und leitet Anfragen an den Server weiter.

- **Applets:** Applets werden in der *Java Runtime Environment* auf dem Client ausgeführt und nicht im Browser, obwohl sie in die dort dargestellte Seite eingebettet sein können.
- **Client-Anwendungen:** Es ist möglich ohne die Verwendung eines Webservers direkt aus einer Anwendung heraus auf Anwendungen innerhalb des Application Servers zuzugreifen. Der Vorteil liegt in den erweiterten Möglichkeiten der Oberflächengestaltung mittels einer GUI-API, wie z.B. AWT oder SWT, im Gegensatz zu den beschränkten Möglichkeiten einer Markup- oder Auszeichnungssprache. Auszeichnungssprachen dienen dazu Wörtern, Sätzen oder Abschnitten eines Textes Eigenschaften zuzuordnen. Ein Beispiel für eine häufig verwendete Auszeichnungssprache ist HTML.

Nachfolgend wird nun die konkrete Implementierung eines Web Application Servers vorgestellt.

3.2 IBM WebSphere Application Server

Es gibt einige Hersteller, die Web Application Server im Rahmen ihrer Softwareproduktpalette anbieten (IBM, Oracle, SAP, Bea, JBoss-Projekt, ...). Der in dieser Arbeit verwendete Web Application Server *WebSphere Application Server*, im Folgenden WAS genannt, stammt von IBM und liegt aktuell in Version 6.1 vor. Er erfüllt die Java EE-Spezifikation V1.4 und verwendet das Java SE SDK V5.0 von IBM. Abbildung 3.7, S. 17, zeigt den Aufbau einer WebSphere-Installation.

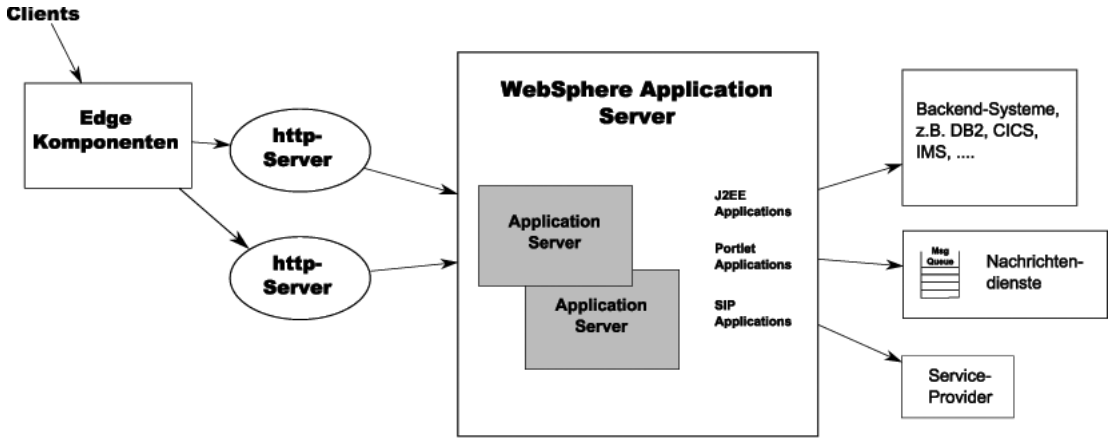


Abbildung 3.7: Konzeptioneller Aufbau einer WebSphere-Installation

Das Modell setzt das 3-Tier-Architekturmodell um, allerdings wird hier der http-Server als Tier 1 gezählt und nicht die Clients ([uFA+06]).

Der WAS unterstützt drei Applikationsformen ([Sad06]). Zuerst die für diese Arbeit relevanten Java EE-Applikationen, in Spezifikation V1.4 noch J2EE-Anwendungen genannt. Darüber hinaus die hier nicht weiter behandelten Portlet- und SIP-Anwendungen. Für mehr Informationen über letztere sei auf [Sad06] verwiesen.

Der WAS ist auf einer Vielzahl von Plattformen verfügbar:

- **Distributed Platforms:** Hierzu zählen IBM AIX, HP-UX, Linux, Solaris, und Microsoft Windows.

- **z/OS:** Das IBM-Betriebssystem für den Mainframe.
- **System i:** Eine weitere IBM-Plattform, kleiner als der Mainframe und mit i5/OS als Betriebssystem. Auf diese Plattform wird in dieser Arbeit nicht eingegangen.

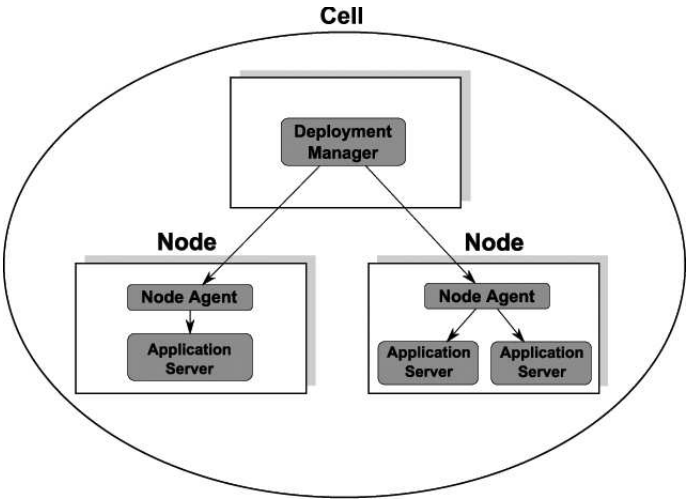


Abbildung 3.8: Verteilte Topologie des WebSphere Application Server

Die Installation eines WAS wird logisch in *Cells* und *Nodes* unterteilt, mit *Deployment Managern* und *Node Agents* als Kontrollinstanzen der jeweiligen Cell bzw. Node (vgl. Abbildung 3.8).

Über den Deployment Manager können zentral die untergeordneten Nodes und Server administriert werden. Technisch betrachtet ist er eine Serverinstanz in einer eigenen JVM, in der keine anderen Anwendungen ablaufen. Eine Cell fasst einen oder mehrere Nodes logisch zu einer Domäne zusammen.

In einem Node werden ein oder mehrere Application Server zusammengefasst und über den Node-Agent zentral verwaltet. In einer Base-Installation wird immer auch ein Node-Agent installiert. Einzelserver sind somit *ND-ready* und können bei Bedarf in eine verteilte Installation aufgenommen werden. Nur auf dem Server mit dem Deployment-Manager muss in diesem Fall ein Network Deployment-Paket installiert werden. Ein Node kann mehrere Application Server enthalten, er erstreckt sich aber nie über mehrere physikalische Maschinen. Das geht nur mit mehreren Nodes, die dann eine Cell bilden, s. Abbildung 3.8 ([Sad06]). Unter z/OS müssen die zu einer Cell zusammengefassten Nodes teil eines Sysplexes sein. Auf anderen Betriebssystemen kann sich eine Cell aus verschiedenen Nodes auf heterogenen Maschinen zusammensetzen.

Die WebSphere-Komponenten

Der WebSphere Application Server setzt sich aus vielen Komponenten zusammen, die interagieren und für den Betrieb des Anwendungsservers notwendig sind (vgl. Abbildung 3.9, S. 19). Sie werden nun der Reihe nach vorgestellt ([BED+04]).

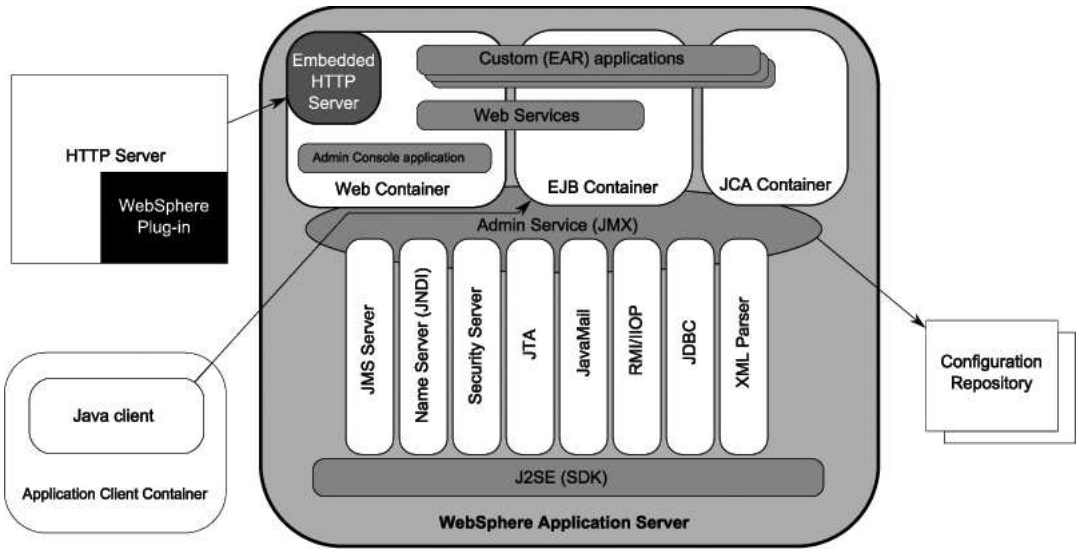


Abbildung 3.9: Komponenten eines WebSphere Application Servers ([BED+04])

Wie schon erwähnt, werden Servlets und JSPs im **Web Container** ausgeführt. Im Web Container läuft auch die **Admin Console Application** (Administrationskonsole). Die Administrationskonsole ist eine Anwendung, die über einen Browser aufgerufen werden kann und über die man, je nach Installationsart, den einzelnen Application Server, die Nodes oder die gesamte Cell administrieren und Änderungen an den Servereinstellungen vornehmen kann.

Die Einstellungen werden im **Configuration Repository** im XML-Format gespeichert. In einer Cell werden cellweite Änderungen über den Deployment Manager vorgenommen. Durch einen Synchronisationsvorgang werden Änderungen an die darunter liegenden Nodes weiter gegeben und in deren lokalem Configuration Repository abgespeichert. Bei einem Ausfall des Deployment Managers können so die Nodes und die unter ihnen liegenden Server weiterhin auf die Konfiguration zugreifen. Bei der Synchronisation werden die Einstellungen der verschiedenen Level kombiniert. Ist auf verschiedenen Leveln (Cell, Node, Server) dieselbe Einstellungen definiert, so hat nur der Wert auf dem niedrigsten Level Bedeutung und überlagert alle vorhergehenden. Werden zum Beispiel auf Nodelevel ein Nameserver und eine Datenbank und auf Serverlevel eine Datenbank definiert, so gilt für den Application Server im Endeffekt die Nameserver-Einstellung des Nodes und die Datenbank-Einstellung des Servers ([BED+04]).

Der **EJB Container** ist die Laufzeitumgebung für Enterprise JavaBeans. Clients für den EJB Container sind entweder der Web Container, im Falle eines Zugriffs über JSPs und Servlets, eine Java-Anwendung oder andere EJBs. Wird entfernt auf ein EJB zugegriffen, d.h. das angefragte EJB ist in einem anderen Server instanziert, so greift das EJB über das RMI/IIOP-Protokoll auf das entfernte EJB zu. Der Container übernimmt dabei Implementierungsdetails wie Transaktionssicherheit, Datenpersistenz, Sicherheit, Caching, etc (vgl. S. 14).

Die *Java EE Connector Architecture*, kurz **JCA**, ist ein Architekturstandard, um auf *Enterprise Information Systems* zuzugreifen. Das sind zum Beispiel Transaktionsserver, Datenbanken oder Geschäftsplanungssysteme. Dabei stellen die Hersteller dieser *EIS* eine standardkonforme JCA-Schnittstelle in ihren Produkten bereit. Mit JCA wird Java EE von einer Middleware-Plattform zu

einer Middleware- und Integrationsplattform ([Ham05]).

Das *Java Transaction Interface*, kurz **JTA**, stellt Schnittstellen zu Transaktionsdiensten zur Verfügung. Der *Java Transaction Service* ist eine Implementierung eines Transaktionsmanagers und wird von IBM im WAS-Paket bereits mitgeliefert.

JDBC ist eine Java-API, um auf relationale Datenbanken oder auch Java Message Services zuzugreifen. Dabei kann *Connection Pooling* verwendet werden, um ressourcenschonend zu arbeiten. Connection Pooling bedeutet, dass in einem Pool eine Anzahl von Verbindungen zur Datenbank vorgehalten und bei Bedarf wiederverwendet wird, da das Öffnen und Schließen einer solchen Verbindung sehr aufwändig ist ([Sad06]).

Der *Java Message Service*, kurz **JMS**, ist notwendig für den Betrieb von Message-Driven Beans (vgl. S.15). Seit Version 5.0 wird im WAS ein JMS mitgeliefert, es kann aber auch die Implementierung eines anderen Messageproviders installiert werden. In der Basisausgabe des WAS (base package) läuft der JMS-Server in einer JVM mit dem Web- und EJB-Container, in der ND-Edition hingegen in einer eigenen JVM.

Das *Java Naming and Directory Interface*, kurz **JNDI**, ist eine API, um Nameserver anzusprechen. In jedem WAS ist ein JNDI-konformer Nameserver enthalten. Der dazugehörige Namespace gilt „cell"-weit und ist baumartig strukturiert. Er ist von tragender Bedeutung für den Betrieb des Servers, da alle EJBs und sonstigen Java EE-Ressourcen, wie etwa JMS, JDBC, J2C, URL und JavaMail, dort unter einem Namen registriert sind. Die Ressourcen, bzw. ihre *bindings* sind dabei im Configuration Repository hinterlegt und können so unabhängig von der Implementierung einer Anwendung verändert werden. Über den Nameserver finden sich dann zum Beispiel EJBs oder auch EJBs die Datenbank.

Ist die *Global Security* im WAS aktiviert, so übernimmt der **Security Service** Authentifizierung und Authorisierung von Benutzern. Dabei baut der Security Service auf bestehende Sicherheitsmechanismen des darunter liegenden Betriebssystems auf.

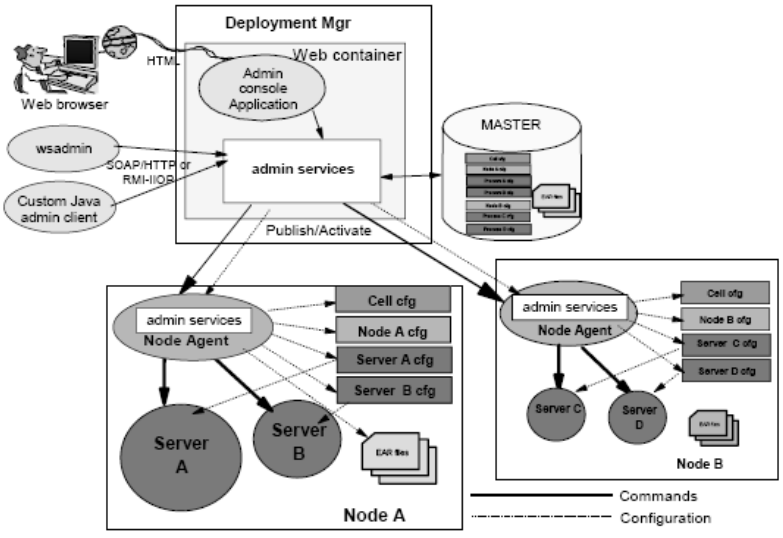


Abbildung 3.10: Hirarchie der Adminservices ([uFA+06])

Der **Admin Service** oder auch *Administrative Dienst* basiert auf der Idee eines Proxys. Also eines zwischen den Services vermittelnden Dienstprogrammes, der auf dem *JMX*-Framework basiert. Die *Java Management Extensions* stellen eine API für den entfernten Zugriff auf Ressourcen, Services, Anwendungen oder auch, wie in diesem Fall, die Konfigurationen im Configuration Repository zur Verfügung. Administrationsclients greifen über verschiedene Protokolle (SOAP/HTTP, RMI/IIOP) auf den Proxy zu (vgl. Abbildung 3.10). Dieser überträgt die vorgenommenen Einstellungen ins Configuration Repository und leitet sie ggf. an die Admin Services der weiteren Nodes/Server weiter.

Es gibt drei Möglichkeiten einen WAS zu administrieren: Die Adminkonsole, die in einem Webbrowser dargestellt wird, wsadmin-Scripting, für automatisierte Konfiguration über (zeitgesteuerte) Skripts und Kommandozeilen-Tools.

Die **Custom (EAR) Applications** sind die (Geschäfts-)Anwendungen, die innerhalb des WAS ablaufen. Sie werden im *Enterprise Application Archive*-Format auf dem WAS installiert und laufen unter Verwendung der vorgestellten Dienste ab.

Im Gegensatz zu den vorhergehenden Diensten sind **Webservices** keine separat implementierte Einheit innerhalb des WAS. Webservices werden verwendet, um den Java EE-Anwendungen lose gekoppelten Zugriff auf heterogene, externe Systeme zu ermöglichen. WAS bietet dabei vollen Support für SOAP-basierten Zugriff auf Webservices mitsamt UDDI und einem **XML-Parser**, zur effizienten Verarbeitung der dabei ausgetauschten SOAP-Nachrichten. SOAP stand früher für *Simple Object Access Protocol*, aber seit Version 1.2 wird der Name weggelassen und das Akronym steht für sich selbst ([Ham05]). SOAP baut auf Transportprotokollen wie HTTP oder SMTP auf und nutzt diese zur Übertragung von Daten im XML-Format. Der *Universal Description, Discovery and Integration*-Service ist der für Webservices verwendete Namens- und Verzeichnisdienst. Als Schnittstellensprache für Webservices wird die *Webservices Description Language*, kurz *WSDL*, verwendet. In WSDL wird die Schnittstelle des Webservices beschrieben und veröffentlicht. WSDL verwendet ebenfalls das XML-Format zur Datenübertragung. Für weitergehende Informationen und die zur Verfügung APIs zu Webservices im WAS sei auf [Ham05] und [Sad06] verwiesen.

Die **JavaMail API** ist ein Framework zur Verwendung von E-Mail-Funktionen in den Anwendungen. Der Mailverteiler, der in WebSphere mitgeliefert wird, unterstützt die gängigen Protokolle wie das *Simple Mail Transfer Protocol*, kurz *SMTP*, das *Internet Message Access Protocol*, kurz *IMAP*, und das *Post Office Protocol Version 3*, kurz *POP3*.

Das **WebSphere Plugin** für HTTP-Server wird als Plugin in den verwendeten Webserver installiert, welcher dem WAS vorgeschaltet ist. IBM stellt das Plugin dabei für eine Reihe von Webservern unterschiedlichster Hersteller zur Verfügung. Das Plugin im Webserver entscheidet bei jedem Request, ob der Request an den WebSphere-Server weitergeleitet wird oder vom Webserver, im Falle statischen Inhalts, direkt beantwortet werden kann. Im ersten Fall wird der Request an den im Webcontainer eingebetteten HTTP-Server in der WAS-Instanz weiter geleitet und von hier aus dann JSPs und Servlets aufgerufen, die wiederum EJBs verwenden können, um Backendsysteme anzusprechen. In einem verteilten Szenario findet im HTTP-Plugin des Webservers ein erstes Verteilen der Last auf die verschiedenen Server statt [BED+04]. Es ist möglich den in den Webcontainer eingebetteten HTTP-Server direkt anzusprechen. Dies wird aber nur für Testzwecke empfohlen [BED+04].

Der Kontrollfluss einer Java EE-Anwendung

Ein einfaches Beispiel soll zeigen, wie der Kontrollfluss einer Anwendung durch den Server vom Request des Clients bis zur Beantwortung des Requests durch den Webserver verläuft ([Sad06]).

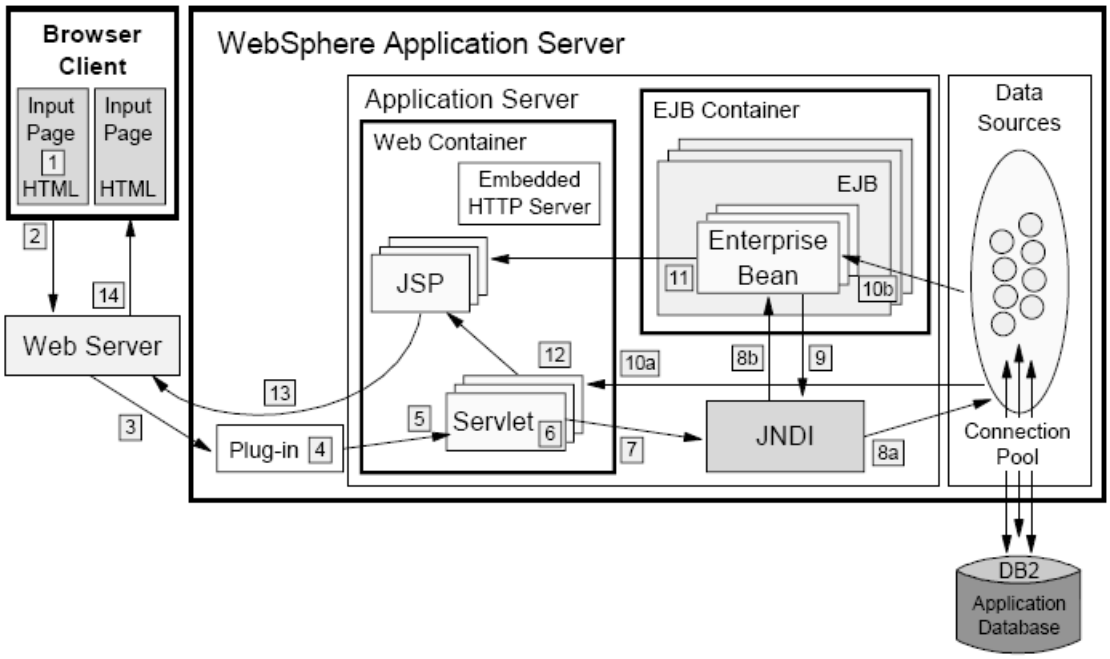


Abbildung 3.11: Ablauf einer Java EE-Anwendung

1 Der Client ruft eine Seite im Browser auf.

2 Durch das Internet wird die Anfrage zum Webserver geroutet.

3 Der Webserver leitet die Anfrage an das HTTP-Plugin des WAS weiter.

4 Das Webserver-Plugin wertet die angeforderte URL aus und prüft, ob diese einem der definierten virtuellen Hosts entspricht. Ein virtueller Host versetzt den WAS in die Lage, in einer physikalischen Installation mehrere unabhängig konfigurierte und administrierte Applikationen zu verwalten, sprich er ist nach außen hin über mehrere URLs ansprechbar. Wird kein virtueller Host gefunden, welcher der angeforderten URL entspricht, wird ein 404-Fehler generiert und an den Client zurück gegeben.

5 Das Plugin öffnet eine Verbindung zum Webcontainer und leitet den Request an diesen weiter. Der Webcontainer prüft die angeforderte URL und versendet den Request an das entsprechende Servlet.

6 Für den Fall, dass die Servletklasse im Moment nicht geladen ist, startet der *Class-Loader* das entsprechende Servlet.

7 Mittels einem JNDI-Aufruf wird nach benötigten EJBs oder Datenquellen gesucht.

8 Je nachdem, ob ein EJB oder Daten aus der Datenbank benötigt werden, wird das Servlet vom JNDI-Service an

- **a** die Datenbank weiter geleitet. Dabei erhält es eine Verbindung aus dem Verbindungspool, um seine Anfragen zu stellen.
- **b** den EJB-Container weiter geleitet, der das benötigte EJB instanziiert.

9 Benötigt das EJB seinerseits eine Verbindung zur Datenbank, sucht es diese wiederum über den JDNI-Service.

10 Die SQL-Anfrage wird gestellt und die resultierenden Daten entweder

- **a** an das anfragende Servlet zurück gegeben oder
- **b** an das anfragende EJB.

11 Die DataBeans werden erstellt und an die JSPs weiter geleitet.

12 Das Servlet leitet seine Daten ebenfalls an die JSPs weiter.

13 Die JSP generiert mit Hilfe der erhaltenen Daten den HTML-Code und leitet ihn an den Webserver zurück.

14 Der Webserver sendet die mit Hilfe von Servlets, EJBs, Datenbankanfragen und JSPs erstellte HTML-Seite an den anfragenden Browser zurück. Dies ist der letztendliche Response in der Kette von Request-Response-Vorgängen.

Nachdem nun der Einsatz von Application Servern motiviert und die technischen Grundlagen allgemein und an einem konkreten Beispiel vorgestellt wurden, wird im nächsten Kapitel auf die Unterschiede zwischen den in den Tests verwendeten WebSphere Application Server für z/OS und WebSphere Application Server für andere Systeme eingegangen.

4 WebSphere Application Server - Unterschiede zwischen z/OS und Linux on IBM System z

Java EE-Anwendungen übernehmen die Java-Eigenschaft „Write once, run anywhere“. Sie sind praktisch plattformunabhängig und für den Endanwender bleiben Unterschiede der Host-Plattformen transparent. Es gibt für den Betrieb eines WebSphere Application Servers aber signifikante Unterschiede aus technischer Sicht. Diese Unterschiede sind vor allem für den Administrator des Servers und unter Umständen für den Entwickler der Anwendung von Bedeutung.

Die Hauptkomponenten eines WebSphere Application Servers sind für alle Systeme dieselben und aus vorhergehendem Kapitel bekannt. Abbildung 4.1 ([IBM08b]) zeigt diese nochmals:

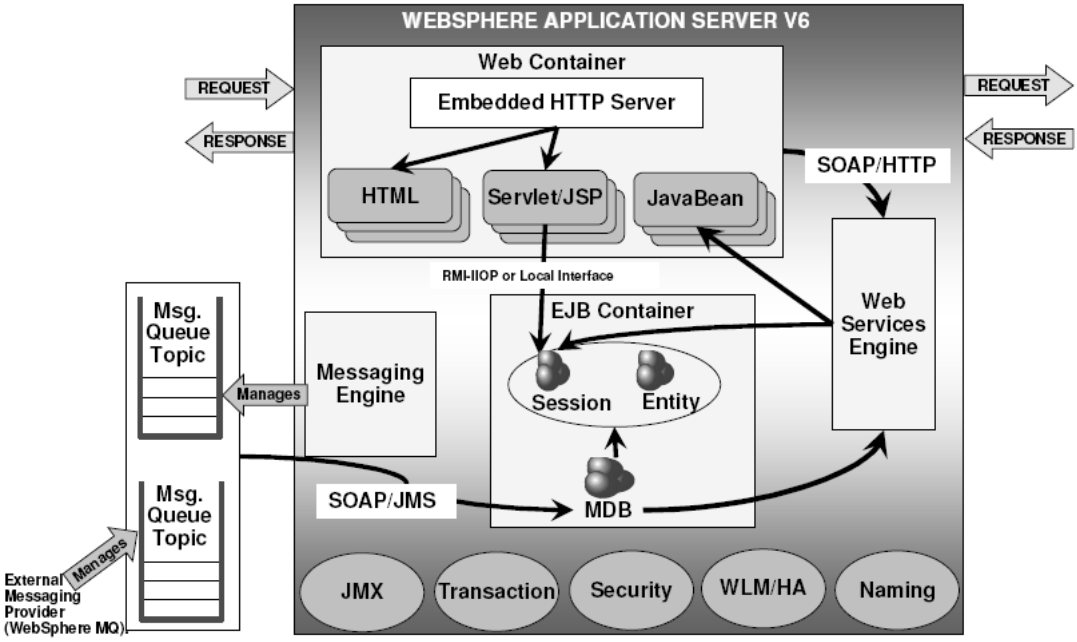


Abbildung 4.1: Komponenten eines WebSphere Application Server V6.

Diese Funktionen bietet der WebSphere Application Server auf allen Plattformen. Im Folgenden wird der Unterschied zwischen dem WebSphere Application Server für z/OS, nachfolgend WebSphere z/OS genannt, und den WebSphere Application Servern für andere Plattformen, nachfolgend WebSphere Distributed genannt, aufgezeigt.

4.1 Das WAS-Prozessmodell unter z/OS

Auf Nicht-z/OS-Systemen läuft der Server aus Abbildung 4.1 in einem einzigen Prozess oder auch Adressraum. Diesen Adressraum bildet eine Java Virtual Machine, die alle Komponenten des WebSphere Distributed enthält. Auf z/OS besteht der WebSphere z/OS aus mehreren Prozessen oder Adressräumen. Diese werden nun nachfolgend erläutert.

4.1.1 Die Control Region

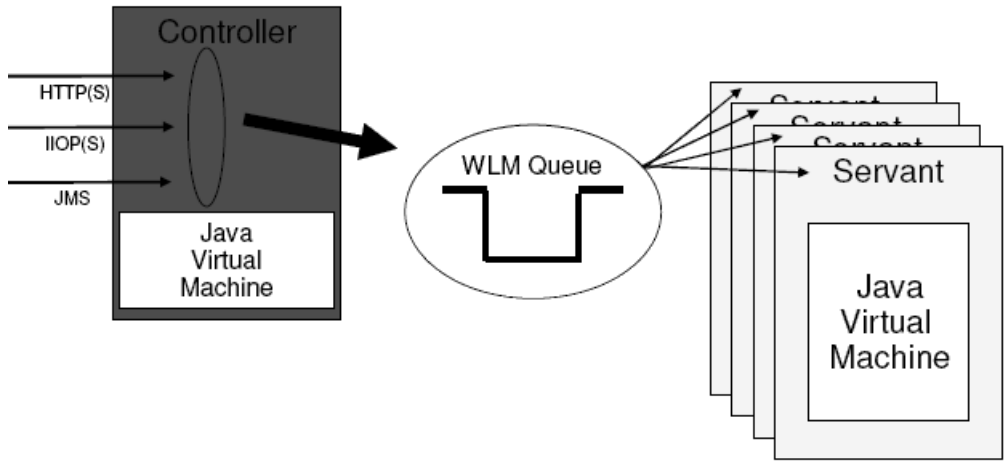


Abbildung 4.2: Control und Servant Regions eines WebSphere z/OS ([IBM08b]).

Control und Servant Regions laufen je in einer Java Virtual Machine. Die *Control Region* ist der Kommunikationsendpunkt des Servers. Hier kommen HTTP-, IIOP- und JMS-Requests an und werden unter Benutzung des *Workload-Management*-Subsystems von z/OS, kurz *WLM*, an die Servant Regions weiter geleitet. Den Servant Regions ist eine Queue vorgelagert, die von WLM verwaltet wird und aus denen sich die Servant Regions die Requests zur Bearbeitung abholen. In der Control Region läuft kein Anwendungscode. Der Anwendungscode läuft in den Servant Regions.

Workload Management

Um die Funktion der Control Region zu verstehen, muss zuerst WLM erklärt werden. WLM ist ein z/OS-Subsystem, das die zur Verfügung stehenden Ressourcen einer z/OS-Instanz verwaltet. Auf einem z/OS laufen für gewöhnlich viele Anwendungen, wie Transaktionsmanager, Datenbanken, Batch-Jobs und Anwendungsserver, parallel. Der Zugriff auf die zur Verfügung stehenden Hardwareressourcen durch diesen *Mixed Workload* wird von WLM verwaltet. Dabei sammelt WLM

stetig Informationen über den Zustand des Systems und ordnet Ressourcen so zu, dass folgende Punkte optimiert werden ([BED+04]):

- **Paging**: Das Ein- und Auslagern von Seiten im Hauptspeicher auf bzw. von nichtflüchtigen Speichermedien, wie Festplatten.
- **Dispatching**: Der Zugriff von Prozessen auf den Prozessor.
- **Swapping**: Das Ein- und Auslagern ganzer Prozesse auf Festplatte.
- **I/O-Prioritäten**: Der Zugriff auf persistente Datenträger oder das Netzwerk.
- **Anzahl der Adressräume**, die Berechnungen durchführen.

WLM verwaltet diese Vorgänge für die ablaufenden Prozesse dynamisch.

Wie funktioniert WLM?

Man kann sich WLM als eine Art Abkommen zwischen Installation (Benutzer) und Betriebssystem vorstellen. In der Literatur wird Workload-Management mit der Verkehrsregulation im Straßenverkehr verglichen ([uAD+07]).
Die Zeit, die ein Fahrzeug vom Start zum Ziel benötigt, wird beeinflusst durch Geschwindigkeitsbeschränkungen (Rechenleistung des Systems), die Anzahl der anderen Verkehrsteilnehmer (Anzahl der um Ressourcen konkurrierenden Prozesse) und dem Fluss durch Knotenpunkte wie Kreuzungen oder Auffahrten (Schnittstellen zu den Hardwareressourcen, wie Kanalpfade, und die Ressourcen eines Systems, wie Festplatten). Die Geschwindigkeit, mit denen sich ein Fahrzeug bewegt, ist durch Geschwindigkeitsbeschränkungen vorhersagbar (Theoretisch mögliche, maximale Rechenleistung des Systems). An Knotenpunkten treten Verzögerungen auf, deren Dauer von der Anzahl der anderen Verkehrsteilnehmer abhängt (Wie viele Prozesse benötigen die Ressource außerdem noch?). Auf diesen Informationen basierend kann nun ein Verwaltungsmechanismus erstellt werden, der den Verkehr so reguliert, dass für alle Teilnehmer das bestmögliche Zeitverhalten entsteht. So gibt es im Straßenverkehr spezielle Fahrstreifen für Busse und Taxis. Notfallfahrzeuge, wie Krankenwagen und Feuerwehrautos, können durch akustische und optische Signale den anderen Verkehrsteilnehmer klar machen, dass sie höhere Priorität und höhere Durchfahrtsrechte besitzen.

Diese Konzepte sind in WLM ähnlich. Alles was es dazu braucht ist eine Technik um folgende Aufgaben zu lösen([uAD+07]):

- Im System ablaufende Last identifizieren.
- Die Zeit zu messen, die diese Last benötigt.
- Verzögerungspunkte identifizieren, sogenannte *Bottlenecks*.

In z/OS wird die Identifikation ankommender Last durch die Middleware, hier die Control Region des WebSphere z/OS, und das Betriebssystem übernommen. Die Control Region meldet sich zu diesem Zweck als *Work Manager* beim WLM an. Die Rolle des Work Managers kann auch andere Middleware, wie zum Beispiel eine Datenbank oder ein Transaktionsmanager übernehmen. Im Zusammenspiel mit dem Betriebssystem teilt der Work Manager WLM mit, wenn neue

Last in das System kommt und es wieder verlässt. Dabei bietet WLM Mechanismen an, um Last in verschiedene Klassen zu unterteilen. Dieser Vorgang wird als *Last-Klassifizierung* (engl. *work classification*) bezeichnet. Die einzelnen Arbeitseinheiten (engl. *Unit of Work*, kurz *UoW*) stehen im Wettbewerb zueinander um die zur Verfügung stehenden Ressourcen, wie Prozessoren, I/O-Geräte, Speicher, usw.

WLM überwacht die Ressourcen und protokolliert dabei, welche Arbeitseinheit gerade welche Ressource verwendet oder benötigt. Basierend auf diesen Beobachtungen und unter Verwendung der in der Installation vereinbarten Ziele werden die Ressourcen den einzelnen Verbrauchern zugeteilt. Der WebSphere-Server ist nicht die einzige Anwendung, die WLM verwendet, aber die für diese Arbeit einzig relevante. WLM verwaltet die Hardwareressourcen für alle in der z/OS-Instanz ablaufende Last.

Mit der Klassifizierung der Last wird jeder Arbeitseinheit eine Zielvorgabe zugeordnet. Diese Zielvorgaben sind in so genannten *Serviceklassen* (engl. *service classes*) definiert. Es gibt dabei drei Arten von Zielvorgaben:

Durchschnittliche Antwortzeit (*average response time)*

Hier wird als Ziel die Zeit angegeben, die eine Arbeitseinheit im System von ihrem Eintritt bis zur Beendigung der auszuführenden Aufgabe benötigen darf. Ein mögliches Ziel ist hierbei zum Beispiel: *Antwortzeit für alle Arbeitseinheiten dieser Klasse kleiner 0,5 Sekunden*.

Es kann hier ein Problem entstehen, falls einzelne Arbeitseinheiten mehr Zeit benötigen, als vorgesehen. WLM würde dann Maßnahmen ergreifen, um dies zu verhindern. Es ist aber völlig vertretbar, dass z.B. 10% der Arbeitseinheiten länger brauchen als die vereinbarten 0,5 Sekunden. Daher ist es besser anstelle der starren durchschnittlichen Antwortzeit die flexible perzentile Antwortzeit (engl. *percentile response time*) zu benutzen. Dabei muss nur ein bestimmter Prozentsatz der Arbeitseinheiten die vereinbarte Antwortzeit einhalten, also zum Beispiel: *Bearbeitung von 90% der Arbeitseinheiten in weniger als 0,5 Sekunden* ([uAD+07]).

Ausführungsgeschwindigkeit (engl. *velocity*)

Velocity ist ein Maß, das angibt, wie schnell Last abgearbeitet wird. Dabei entspricht eine Ausführungsgeschwindigkeit von 100% der Tatsache, dass die Arbeitseinheit zu jedem Zeitpunkt alle benötigten Ressourcen sofort erhalten hat und nicht durch Warten auf Ressourcen verzögert wurde. Rechnerische Grundlage dafür ist die Formel:

$$Velocity = \frac{Using\ samples}{Using\ samples + Delay\ samples}$$

Ein Sample ist dabei der Zustand der Arbeitseinheit zum Zeitpunkt der Beobachtung durch das System. Zu diesem Zeitpunkt ist eine Arbeitseinheit entweder in dem Zustand, dass sie eine benötigte Ressource benutzt (using) oder auf eine benötigte Ressource wartet (delayed).

Rechenbeispiel: Eine Arbeitseinheit wird 16 mal in ihrem Verlauf beobachtet. Dabei ist sie vier mal im Zustand *using* und zwölf mal im Zustand *delayed*. Rechnerisch ergibt das eine Ausführungsgeschwindigkeit von 25% ($\frac{4}{4+12} = 0,25$).

Anders ausgedrückt kann man sagen, dass eine Arbeitseinheit, die bei perfekter Ausführung vier Sekunden benötigen würde, aber in der tatsächlichen Ausführung 16 Sekunden benötigt, eine Ausführungsgeschwindigkeit von 25% besitzt.

Willkürlich (engl. *discretionary*)

Last mit dieser Zielvereinbarung wird ausgeführt, wenn Ressourcen nicht durch andere Arbeit belegt sind, und hat keine spezielle Zielvereinbarung und keine Priorität (s.u.).

In einer Serviceklasse werden noch weitere Parameter definiert, von denen manche optional sind:

- **Priorität** (engl. *importance*): Der wichtigste Parameter einer Serviceklasse. Hier wird ein Wert zwischen eins und fünf angegeben, wobei 5 der niedrigsten Priorität entspricht und 1 der höchsten. Die Priorität gibt an, wie wichtig es ist, dass eine Arbeitseinheit ihre Ziele erreicht. Schafft es der WLM nicht, alle vereinbarten Ziele zu erreichen, werden die Ressourcen der Priorität nach zugeteilt. Werden alle Ziele erreicht, ist dieser Wert bedeutungslos.

- **Laufzeitperiode** (engl. *duration*): Die Laufzeitperiode gibt die Gültigkeitsdauer einer Serviceklasse für eine Arbeitseinheit an. Eine Serviceklasse kann in mehrere Perioden (engl. *periods*) unterteilt sein. Hat eine Arbeitseinheit ihre Ziel nicht innerhalb einer bestimmten Zeitspanne erreicht, altert sie und rutscht in die nächste Laufzeitperiode. So wird verhindert, dass lang laufende Arbeitseinheiten zu viele Ressourcen beanspruchen, weil sie eine zu hohe Priorität erhalten.

- (*Optional*) **Ressourcengruppe** (engl. *resource group*): Die Ressourcengruppe gibt ein Minimum und ein Maximum der *CPU Service Units* pro Sekunde an. IBM empfiehlt, diese Konfigurationsmöglichkeit nicht zu nutzen ([BED+04]).

- (*Optional*) **CPU-kritisch** (engl. *cpu critical*): Durch diesen Parameter kann erreicht werden, dass eine Arbeitseinheit die CPU erhält, obwohl höher priorisierte Last die CPU ebenfalls benötigt. Wichtig für CPU-kritische Anwendungen wie z.B. CICS.

- (*Optional*) **Speicherkritisch** (engl. *storage critical*): Es gibt Anwendungen, deren Speicher geschützt werden muss, auch wenn sie lange Zeit nicht aktiv sind und ihr realer Speicher möglicherweise durch Paging ausgelagert werden würde. Es ist aber von Bedeutung, dass der Speicher sofort zur Verfügung steht, sobald diese Anwendungen wieder aktiv werden.

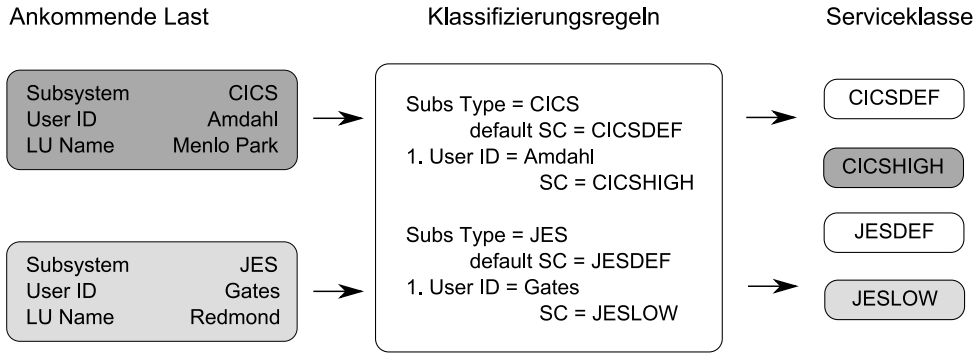


Abbildung 4.3: Klassifizierungsregeln

Die Klassifizierung in die Serviceklassen geschieht anhand von bestimmten Regeln (*classification rules*). Diese Klassifizierungsregeln stellen eine Beziehung zwischen bestimmten Eigenschaften der ankommenden Arbeitseinheiten, z.B. Benutzerkennung, IP-Adresse oder aufgerufenem Subsystem, und Zielvorgaben her. So wird in den Klassifizierungsregeln eine Zuordnung von äußeren Eigenschaften und Serviceklasse vorgenommen (vgl. Abbildung 4.3).

Die möglichen Ziele unterscheiden sich in ihrer rechnerischen Beschreibung, was es schwierig macht, sie zu vergleichen. Zu diesem Zweck benutzt WLM einen *Performance Index*, kurz *PI*. Der Performanceindex gibt dabei nach folgendem Schema an, ob die Ziele der Serviceklassen erreicht werden oder nicht:

- PI $= 1$: Die Zielvorgabe der Serviceklasse wird genau erreicht.
- PI > 1: Die Zielvorgabe der Serviceklasse wird nicht erreicht.
- PI < 1: Die Zielvorgabe der Serviceklasse wird übertroffen.

Die Berechnungsgrundlagen für die unterschiedlichen Zielarten lauten:

$$
\begin{aligned}
\text{Antwortzeit-Ziele: } PI &= \frac{\text{Erreichte Antwortzeit}}{\text{Ziel-Antwortzeit}} \\
\text{Perzentile Antwortzeit-Ziele: } PI &= \frac{\text{Aktuelle Zielerreichung}}{\text{Vorgegebene Zielerreichung}} \\
\text{Ausführungsgeschwindigkeits-Ziele: } PI &= \frac{\text{Ausführungsgeschwindigkeits-Ziel}}{\text{Erreichte Ausführungsgeschwindigkeit}} \\
\text{Willkürliches Ziel: } PI &= 0,81 \text{ (const.)}
\end{aligned}
$$

Die Berechnung des Performanceindexes ist in der Praxis weitaus komplexer, als es die einfachen Formeln vermuten lassen. Für mehr Informationen zum Performanceindex sei an dieser Stelle auf [uAD+07] verwiesen. Durch den PI wird eine rechnerische Basis geschaffen, mit der sich die Zielerreichung der unterschiedlichen Ziele miteinander vergleichen lässt.

Neben den Serviceklassen kann man die im System ankommenden Arbeitseinheiten auch Reportklassen (engl. *report classes*) zuordnen. Diese haben keinen Einfluss auf die Ausführung der Arbeit, sind aber wichtig für die Auswertung der Last. Mit Reportklassen kann man die Last, die schon in Serviceklassen unterteilt wurde, weiter unterteilen. Hat man z.B. Kunden in Paris und Hong-Kong und weist deren Transaktionen derselben Serviceklasse zu, möchte dabei aber wissen, welche Last von jeweils welchem Kundenstandort verursacht wird, so kann man für die Zweigstellen in Paris und Hong-Kong unterschiedliche Reportklassen einrichten. Im Reporting taucht die Last der beiden Standorte dann in verschiedenen Reportklassen auf und ist so unterscheidbar, was rein anhand der Serviceklasse, die hier im Beispiel für beide gleich ist, nicht möglich wäre.

So wie sich verschiedene Reportklassen in einer Serviceklasse zusammenfassen lassen, lassen sich verschiedene Serviceklassen in *Workloads* zusammenfassen. Workloads lassen sich dabei nach Subsystem (CICS, WAS, ...), Anwendungsart (Produktion, Batchbetrieb,...) oder Geschäftszweig (Abteilung, Standort,...) zusammenfassen.

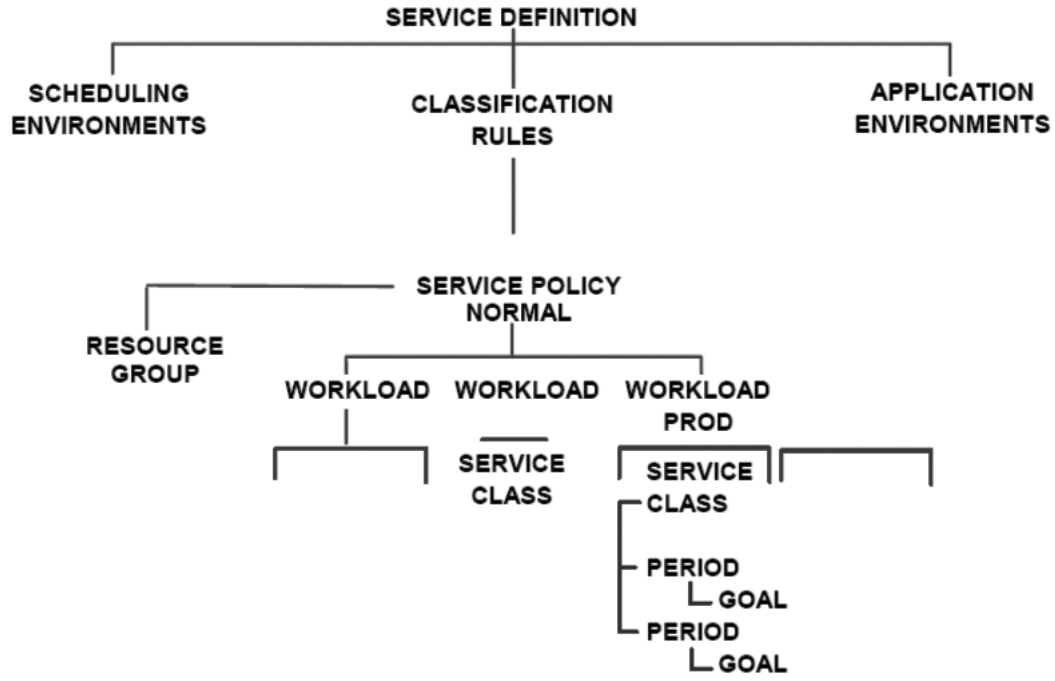


Abbildung 4.4: WLM-Hierarchie im Überblick ([uAD+07])

Abbildung 4.4 zeigt die Hierarchie der WLM-Komponenten. Hier nicht näher besprochene Komponenten sind:

- **Service Definition:** Die Servicedefinition ist ein logischer Container für alle anderen Konfigurationskomponenten des WLM-Subsystems. Nur eine Servicedefinition kann zu einem Zeitpunkt aktiv sein.
- **Service Policy:** Die Servicepolicy fasst Konfigurationen zusammen. Es kann mehrere Servicepolicies geben, aber nur eine ist zu einem Zeitpunkt gültig. Mehrere Policies erlauben es, einfach und schnell die aktive Policy zu wechseln bzw. zu testen.
- **Scheduling Environment:** Die verwaltete Umgebung ist nur für einen Sysplex relevant. Hier stehen die zur Verfügung stehenden Ressourcen zusammen mit ihrem derzeitigen Zustand. So kann man z.B. angeben, dass eine Servermaschine nur zwischen 20:00 Uhr und 6:00 Uhr für einen bestimmten Workload zur Verfügung steht, da tagsüber andere Arbeiten auf der Maschine erledigt werden müssen.

4.1.2 Servant Regions

Ein Request, der die Control Region erreicht, wird also von dieser klassifiziert. Zu jeder konfigurierten Serviceklasse gibt es eine WLM-Queue, also eine Warteschlange für Requests (vgl. Abbildung 4.2, S. 25). An dieser Stelle übernimmt der WLM die Verwaltung des Request und prüft, ob zum bestehenden Request ein passendes *Application Environment* existiert. Ein Application Environment ist eine Menge an Adressräumen gleicher Konfiguration, die für eine bestimmte Klasse an

Transaktionen geeignet sind. Erkennbar sind Server derselben Konfiguration an der gleichen JCL-Start-Prozedur. Die Server sind im Falle von WebSphere z/OS die Servant Regions, in denen der Anwendungscode läuft. Sie sind in Funktion und Aufbau (Web-Container, EJB-Container, etc.) identisch mit den Prozessen im Einprozess-Modell von WAS Distributed ([Yu08]). Die Servant Regions zu einer Control Region müssen sich alle in derselben Systempartition (LPAR) befinden.

Enclaven

Eine Transaktion, die von der Control Region klassifiziert und einer Servant Region zur Bearbeitung übergeben wurde, greift häufig noch auf andere Subsysteme, wie z.B. eine Datenbank, zu. Das bedeutet, die Transaktion durchläuft während ihrer Abarbeitung verschiedene Adressräume. Greift eine WebSphere-Anwendung auf eine Datenbank im selben System zu, hat das folgende Konsequenzen ([uAD+07]):

- Der Transaktionsanteil, der aufgrund einer Anforderung aus der Servant Region eines WebSphere-Servers in einem DB2-Adressraum abgearbeitet wird, verursacht für diesen Adressraum Ressourcenverbräuche.
- Der Transaktionsanteil im DB2-Adressraum wird aufgrund der Priorität dieses verarbeitet. Dies führt dazu, dass verschiedene Clientanfragen an die Datenbank (Client ist hier der WebSphere-Server), die möglicherweise im Clientadressraum unterschiedliche Zielvereinbarungen besitzen, alle im Kontext der Ziele des DB2-Adressraums ablaufen.

Um diese Problematik aufzulösen, führt man sogenannte *Enclaves* ein. Eine Enclave fasst Programmteile, die zu einer Transaktion gehören und über mehrere Adressräume verteilt sind, zu einer Einheit zusammen. Programmteile bezeichnet hier entweder einen TCB oder einen SRB.

An dieser Stelle muss etwas weiter ausgeholt werden. Ein Programm in Ausführung ist aus Betriebssystemsicht unter Unix ein *Prozess* und wird unter z/OS als *Job* bezeichnet. In dem vom Job belegten Speicher befindet sich, vereinfacht ausgedrückt, der Programmcode und die vom Programmcode während seines Ablaufs verwendeten Daten. Ein Programm in Ausführung folgt einem bestimmten Pfad durch den Programmcode. Unter Unix-Systemen wird dieser Programmverlauf als *Thread* bezeichnet. In z/OS-Terminologie spricht man von *Dispatchable Units of Work*, kurz *DuOW* ([uAD+07]), oder auch *Tasks*.

Es können mehrere DUoWs innerhalb eines von einem Job belegten Adressraum ablaufen. z/OS verwaltet dieses parallel ablaufenden Programmteile in so genannten *Kontrollblöcken*, engl. *control blocks*. Es gibt mehrere Arten von Kontrollblöcken ([EOO06]). Die für diese Arbeit wichtigen sind die so genannten *Task Control Blocks*, kurz *TCB* und *Service Request Blocks*, kurz *SRB*. Im TCB verwaltet das z/OS-Betriebssystem Informationen über den laufenden Task, wie zum Beispiel über die von ihm angelegten Adressbereiche und seine Position im Programmcode. Ein SRB ist eine *Serviceroutine*, die von einem anderen Adressraum aufgerufen werden kann. Im Gegensatz zu einem TCB kann ein SRB keine eigenen Speicherbereiche besitzen. Er verwendet während des Ablaufs der von ihm spezifizierten Routine die Adressbereiche anderer TCBs ([EOO06]). Ein SRB ist vergleichbar mit einem *System Call* unter Unix.

Eine Enclave fasst nun diese SRBs und TCBs, die zusammen eine Transaktion ergeben, in einem WLM-Kontext zusammen, damit die Zielvereinbarung, die für diese Transaktion in der Control Region getroffen wurde, auch noch in anderen Adressräumen Gültigkeit besitzt. Die Control

Region erzeugt für ankommende Requests eine Enclave, in deren Kontext die Transaktion dann adressraumübergreifend abläuft. Hier ist es wichtig zu verstehen, dass die Enclave meist andere Zielvereinbarungen besitzt, als der Adressraum der Servant Region, in der die Enclave zum Teil ausgeführt wird. Die Enclave ist aus WLM-Sicht eine als eigenständig zu betrachtende und unter einer Zielvereinbarung verwaltete Einheit. Sie setzt sich aber aus technischer Sicht aus Programmteilen verschiedener Adressräume zusammen.

WLM verwaltet die Zahl der Servant Regions dynamisch, in Abhängigkeit der verfügbaren Ressourcen oder aufgrund von Verzögerung in der WLM-Queue. WLM startet bzw. beendet Servant Regions je nach Bedarf. Da zuvor eine Klassifizierung in Serviceklassen stattgefunden hat und es zu jeder Serviceklasse eine passende WLM-Queue gibt, laufen die Arbeitseinheiten zu jeder Serviceklasse in einem oder mehreren unabhängigen Adressräumen. Diese Isolation erhöht die Sicherheit der Anwendungen.

Die Zahl der Worker-Threads innerhalb der Servant Region lässt sich auf z/OS über einen Parameter je nach Art der zu bearbeitenden Last einstellen. Dabei unterscheidet man zwischen vier Arten von Last und demgemäß auch zwischen vier Einstellungen ([uAD+07]):

- **ISOLATE:** In der Servant Region läuft nur ein einziger Thread zur Bearbeitung von Requests.
- **IOBOUND:** Diese Einstellung wird empfohlen für Anwendungen, die ein Gleichgewicht zwischen prozessorlastigen Berechnungen und entfernten Anfragen besitzen. Die Zahl der eingerichteten Threads wird nach folgender Formel berechnet:

$$MIN(30, MAX(5, (\text{Anzahl der CPUs} \cdot 3)))$$

- **CPUBOUND:** Prozessorlastige Anwendungen profitieren nicht von der Einrichtung vieler Threads, daher gibt es hier, in Abhängigkeit der Anzahl der installierten Prozessoren, nur sehr wenige pro Servant Region:

$$MAX((\text{Anzahl der CPUs} - 1), 3)$$

- **LONGWAIT:** Hier werden mehr Threads aktiviert als bei der Einstellung IOBOUND und diese Konfiguration ist für Anwendungen gedacht, die den größten Teil der Zeit auf die Fertigstellung von Netzwerkoperationen oder entfernten Anfragen warten. Hier werden fest 40 Threads eingerichtet.

An dieser Stelle müssen Administrator und Entwicklung zusammen arbeiten, um diese Konfigurationsmöglichkeit bestmöglich zu nutzen. Für WAS Distributed lässt sich als Äquivalent die minimale und maximale Anzahl an Worker-Threads über die Administrationskonsole einstellen ([uFA+06]).

Abgesehen von den Control und Servant Regions gibt es in einer Serverinstanz auf z/OS noch einen weiteren Adressraum.

4.1.3 Control Region Adjunct

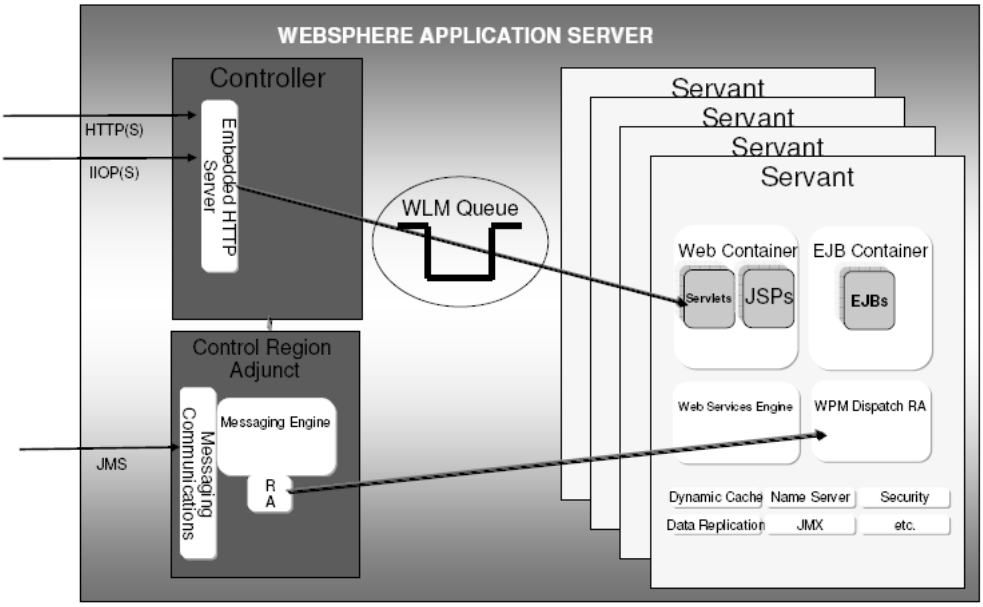


Abbildung 4.5: Die Control Region Adjunct in einer Serverinstanz ([IBM08b]).

Mit Version 6.0.1 wurde die *Control Region Adjunct*, kurz *CRA*, eingeführt. Die CRA ist eine *Message Engine*. In WAS Distributed ist sie in dem Adressraum integriert, der eine Serverinstanz bildet (vgl. Abbildung 4.1, S. 24). Unter z/OS ist sie ein eigener Adressraum und auch nur in z/OS-Installationen zu finden.

Die CRA wird nur benötigt, wenn der WAS Teilnehmer eines *Service Integration Bus*, kurz *SIB*, ist. Der SIB verbindet mehrere Server oder Servercluster zum Zwecke des asynchronen Nachrichtenaustauschs zwischen verschiedenen Anwendungen. Dabei bietet die CRA zwei Schnittstellen an. Eine Schnittstelle geht nach außen über den SIB zu anderen Servern. Die zweite Schnittstelle geht über einen *Resource Adapter* in eine Servant Region zum *WPM Dispatch Resource Adapter*. In der Servant Region werden die nachrichtenbehandelnden Message Driven Beans im EJB-Container letztendlich instanziert.
Basiert der Nachrichtenaustausch, wie in Abbildung 4.5 impliziert, auf dem JMS-Framework (vgl. S. 20), wird der SIB auch als *Messaging Bus* bezeichnet ([IBM08a]). Über den SIB findet z.B. auch die Kommunikation via WebServices statt ([Sad06]).

Die bisherigen Betrachtungen beziehen sich alle auf eine Serverinstanz. Für den Betrieb eines Clusters bei horizontaler und/oder vertikaler Verteilung ergeben sich weitere Unterschiede.

4.2 Verteilter Betrieb von Serverinstanzen

Mit steigender Zahl der gleichzeitigen Benutzer kann eine einzelne Serverinstanz die anfallende Last möglicherweise nicht mehr bewältigen. Aus diesem Grund lassen sich mehrere Server zu

einem *Cluster* zusammenfassen. Die Server in einem Cluster bearbeiten dann gemeinsam die anfallende Nutzerlast. Für den außen stehenden Benutzer erscheint solch ein Servercluster als ein großer Anwendungsserver. Die genaue Topologie bleibt für den Nutzer transparent.

Neben der erhöhten Belastbarkeit hat Clustering auch den Vorteil der Ausfallsicherheit. Fällt in einem Cluster ein Server aus, können die Anfragen von anderen Servern im Cluster bearbeitet werden und es kommt so zu keinem Ausfall des angebotenen Dienstes.

Man kann Cluster auf zwei Arten aufbauen, die sich gegenseitig nicht ausschließen.

4.2.1 Vertikales Clustering

Ein vertikaler Cluster bietet sich für Maschinen der *SMP*-Architektur an (*Symmetric Multiprocessing*). Hierbei werden mehrere Server auf einer Maschine oder auch in einer Partition betrieben. Der Begriff *Partition*) wird ab Seite 40 genauer erläutert. Man erhält hierbei Ausfallsicherheit auf Prozessebene, d.h. der Anwendungsserver bleibt erreichbar, auch wenn einzelne Serverprozesse ausfallen. Fällt in einem vertikalen Cluster die gesamte Maschine aus, ist der Anwendungsserver nicht mehr erreichbar. Bei einer Einzelprozessor-Maschine kann vertikales Clustering durch den Overhead bei Kontextwechseln negative Effekte haben und zu einer Verschlechterung der Leistung führen.

Um die Requests sinnvoll den Mitgliedern eines Clusters zuteilen zu können, benötigt man einen den Servern vorgeschalteten Verteiler, der ankommende Requests gemäß gewisser Regeln an ein bestimmtes Mitglied im Cluster weiterleitet. Im Falle des WebSphere Application Servers sind das das Webserver-Plugin für Anfragen von webbasierten Clients und ein *Object Request Broker*, kurz *ORB*, für Anfragen EJB-basierter Clients.

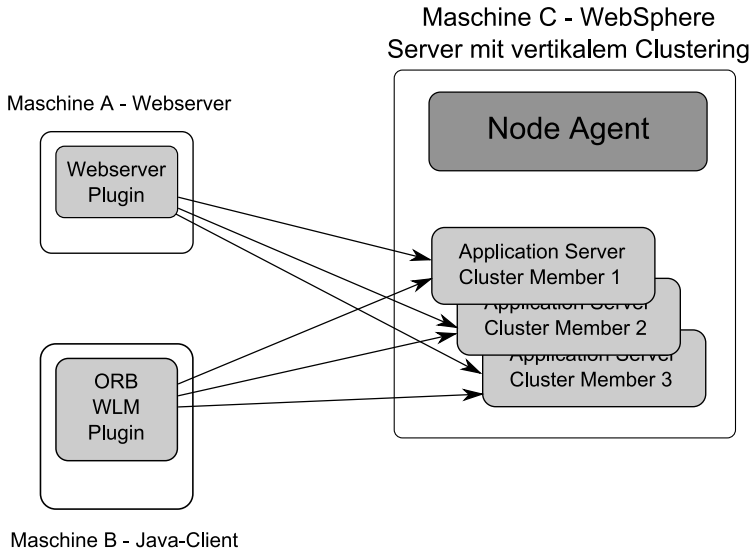


Abbildung 4.6: Ein vertikaler Cluster auf einer Maschine.

Abbildung 4.6 skizziert einen vertikalen Cluster mit den vorgelagerten Verteiler-Plugins.

Die Plugins bekommen hierbei vom Administrator in einer Konfigurationsdatei die Daten der Server zusammen mit einer Gewichtung mitgeteilt. Der Gewichtswert eines Servers gibt an, wie-

viel Requests er erhalten kann. Round Robin oder zufällig wird vom Plugin ein Server ausgewählt, der erhaltene Request an diesen weiter geleitet und der Gewichtswert dieses Servers um Eins erniedrigt. Hat der Gewichtswert eines Servers den Wert Null erreicht, bekommt dieses Clustermitglied keine weiteren Requests mehr. Haben die Gewichtswerte aller Server im Cluster den Wert Null erreicht, werden die Gewichtswerte auf ihren Ursprungswert zurückgesetzt und die Verteilung beginnt von neuem. Dabei ist wichtig, dass der Administrator die Leistungsfähigkeit eines Servers richtig einschätzt und so die anfallende Last optimal unter voller Ressourcenauslastung abgearbeitet wird ([BED+04]).

Auf der Servermaschine läuft ein weiterer Prozess außer den Application Servern, der sogenannte *Node Agent*. Der Node Agent ist die Verwaltungsschnittstelle zu den Application Servern innerhalb eines Nodes (vgl. Kapitel 3, S. 19). Der Node Agent-Prozess ist bei WebSphere z/OS wie auch bei allen WebSphere Distributed vorhanden.

Der Betrieb eines vertikalen Clusters kann im Kostenvergleich zu einem horizontalen Cluster billiger sein ([BED+04]).

4.2.2 Horizontales Clustering

Bei horizontalem Clustering werden die Serverinstanzen auf mehrere Maschinen oder auch Partitionen verteilt. Horizontales Clustering ermöglicht es mit rechenschwächeren Systemen die Leistung eines großen, rechenstarken Systems zu erreichen.

Durch horizontales Clustering erweitert sich die Ausfallsicherheit auf Maschinenebene. Der Leistungsabfall durch den Hardwareausfall einer Maschine kann von anderen Maschinen im Cluster kompensiert werden.

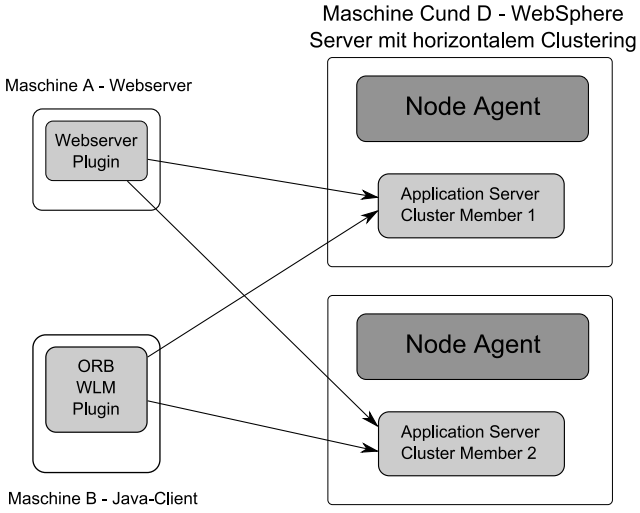


Abbildung 4.7: Ein horizontaler Cluster über mehrere Maschinen hinweg.

Die Nodes eines horizontalen Clusters bilden zusammen eine Cell. Die Server in den Nodes können für eine Cell von einem zentralen Deployment Manager aus verwaltet werden. Dieser Prozess ist eine weitere WebSphere-Serverinstanz, genauer gesagt ein weiterer Node innerhalb des Clusters auf dem nur die Administrationskonsole läuft (vgl. Abbildung 4.8).

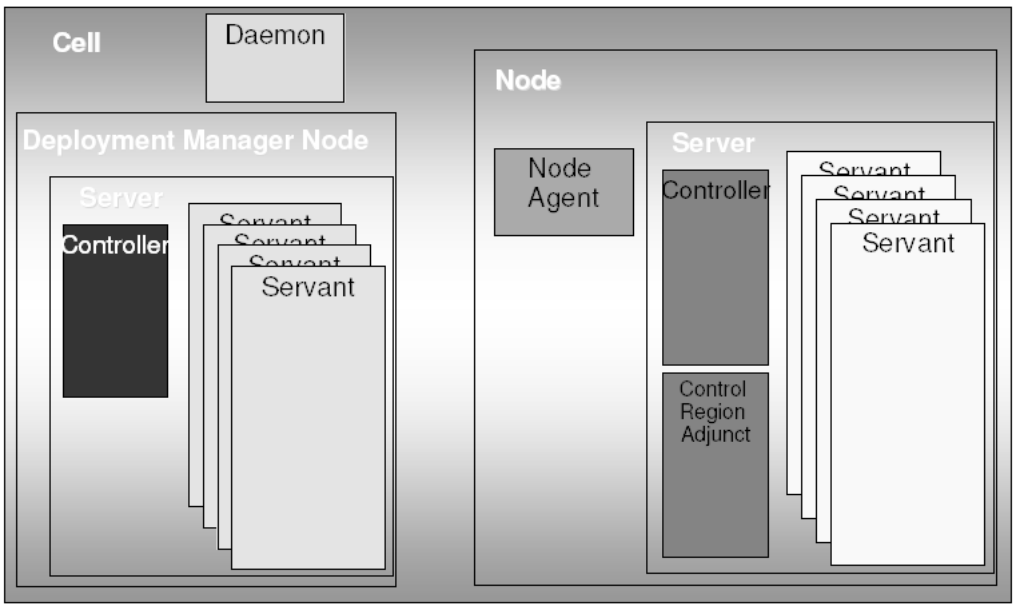


Abbildung 4.8: Übersicht über eine Cell ([IBM08b]).

In Abbildung 4.8 taucht ein weiterer neuer Prozess auf, der nur auf z/OS zu finden ist, der *Daemon*. Der Daemon ist die Implementierung des *CORBA Location Services* und wird benutzt, um RMI/IIOP zu realisieren. Ruft ein Client ein EJB auf, leitet der Daemon unter Verwendung von WLM auf z/OS die Anfrage an einen Server weiter, der dann eine CORBA-Session mit dem Klienten eröffnet. Darauf folgende Anfragen gehen direkt über die erstellte Session ([IBM08a]). Es gibt einen Daemon pro LPAR pro Cell auf System z ([IBM08b]).

Vor- und Nachteile von Clustering

Cluster können eine Notwendigkeit in einer WebSphere-Umgebung sein ([BED+04]):

- Cluster machen eine Anwendung hochverfügbar und bieten, je nach Architektur, vertikale und horizontale Ausfallsicherheit.
- WebSphere-Cluster bieten eine zentrale Konfigurationsstelle, über die sich viele Server zentral verwalten lassen.
- In einem Cluster kann auf gestiegene oder verringerte Anforderungen an die Anwendung reagiert werden.
- Cluster bieten eine gesteigerte Leistung gegenüber Einzelservern.

Cluster besitzen aber auch Nachteile:

- Die Administration eines Clusters ist komplizierter als die eines Einzelservers.
- Die Fehlersuche in einem Cluster ist aufgrund der gesteigerten Komplexität schwieriger.

- Anwendungen, die in einem Cluster ausgeführt werden sollen, müssen aufwändigeren Tests unterzogen werden. Applikationen, die auf Einzelservern korrekt funktionieren, tun dies aufgrund von Seiteneffekten nicht zwangsläufig auch in einem Cluster.

4.2.3 Clustering auf System z

Das bisher über Clustering Gesagte gilt, abgesehen vom Daemon-Prozess, für jede so aufgebaute Infrastruktur einer verteilten WebSphere Application Server-Installation. System z bietet aber im Bereich Clustering von Systemen und Servern einige Vorteile im Vergleich zu Clustering von anderen Einzelsystemen.

Clustering auf System z mit z/OS

Das Clustering von z/OS-Systemen wird als *Parallel Sysplex* (Sysplex von SYStem comPLEX) bezeichnet. Die Architektur eines Parallel Sysplex kann hier nur in aller Kürze besprochen werden. Für mehr Informationen sei auf [EOO06] und [CG+06] verwiesen.

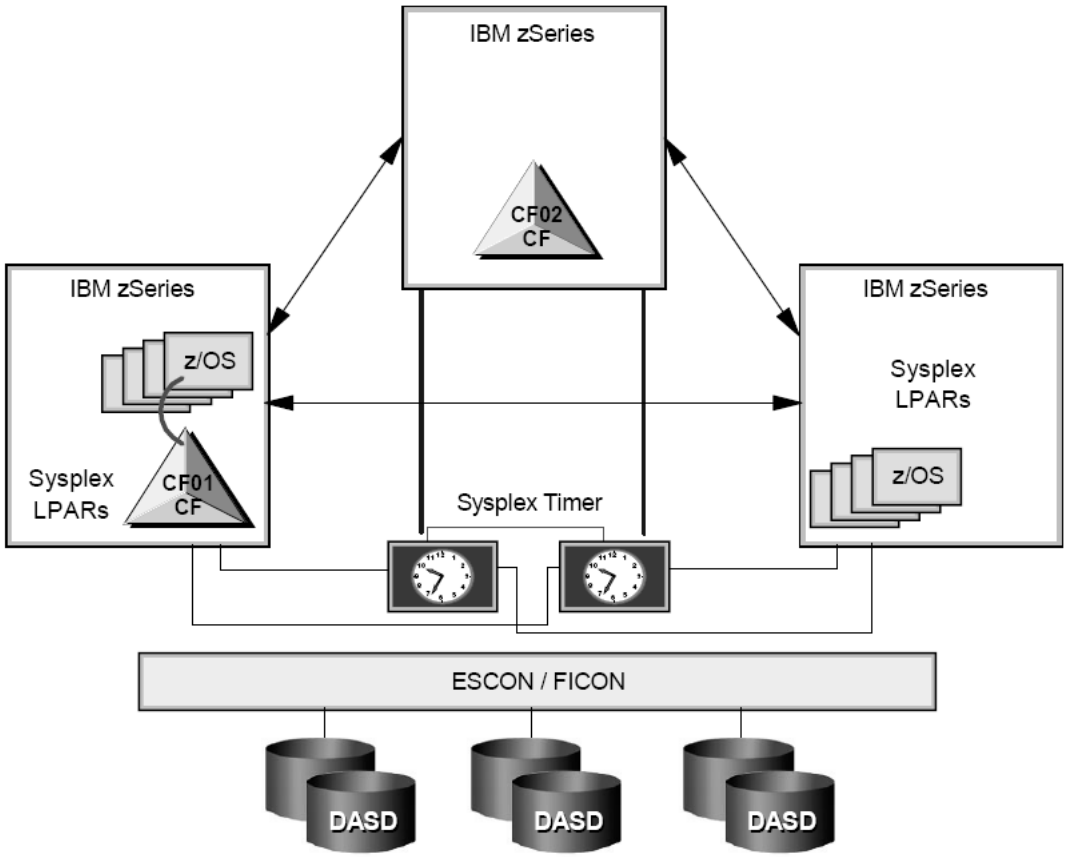


Abbildung 4.9: Überblick über einen Parallel Sysplex ([EOO06]).

Im Zentrum eines Parallel Sysplex steht die *Coupling Facility*, kurz *CF*, von der es zu Redundanzzwecken auch mehrere geben kann. Die CF besitzt einen eigenen Prozessor, mit Speicher

und speziellen I/O-Kanälen, und ein eigenes Betriebssystem. Die CF fungiert als Kommunikationsschnittstelle zwischen den, in einer Stern-Topologie, um sie herum angeordneten System z-Systemen ([vB07]).

Die modernste Generation der System z besitzt in der größten Ausbaustufe 54 Prozessoren, die dem Anwender zur Verfügung stehen ([vB07]). Bis zu 32 System z können in einem Parallel Sysplex zusammengeschaltet werden, d.h. in einem Parallel Sysplex hat man die mögliche Anzahl von 1728 Prozessoren und maximal rund 16TB RAM zur Verfügung.

Die CF erfüllt drei Aufgaben für die im Sysplex verbundenen Systeme:

- Verwalten der Locking-Informationen bei gemeinsamen Zugriff der verschiedenen Systeme auf gleiche Daten.
- Caching von Daten, die über die Systeme verteilt sind.
- Verwaltung von Dateiinformationen für gemeinsam genutzte Daten.

Über einen Sysplex-Timer werden die internen Uhren der Systeme unter Verwendung eines *System Time Protocols* synchronisiert. Alle im Sysplex befindlichen Systeme greifen auf dieselben *Direct Access Storage Devices*, kurz *DASD*, oder auch externen Speicher, wie z.B. Festplatten, zu.

Dabei bietet ein Parallel Sysplex laut Literatur nahezu lineare Skalierung ([CG+06]).

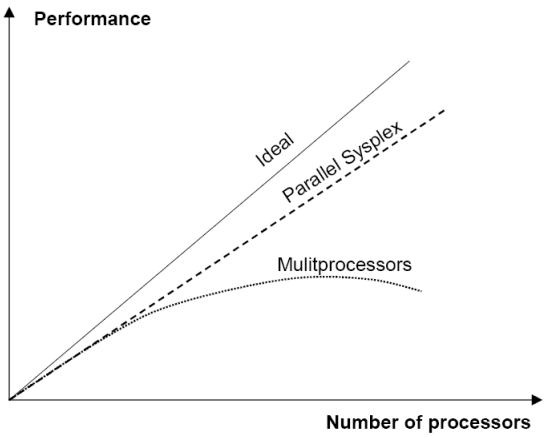


Abbildung 4.10: Skalierung eines Parallel Sysplex ([CG+06]).

Es lassen sich neue Systeme zur Laufzeit und ohne Beeinträchtigung der bereits im Parallel Sysplex laufenden Anwendungen dem Sysplex hinzufügen. Die CF benützt eigens für sie entwickelte Hochgeschwindigkeitsbusse und Protokolle, um mit den zu verwaltenden Systemen Nachrichten auszutauschen. Die Prozessoren der CF besitzen speziell für ihre Aufgabe eingerichtete Hardwareinstruktionen.

Kommen Requests in einem über einen Parallel Sysplex realisierten WebSphere-Cluster an, können die Performance Indexe (vgl. S. 29) der einzelnen Systeme kontrolliert und verglichen werden. Der Request wird dann dynamisch an ein Clustermitglied mit freien Ressourcen weitergeleitet. So werden ausgelastete Systeme geschont und die Gesamtlast optimal auf die Systeme verteilt. Ähnlich wie WLM für ein einzelnes z/OS-System gibt es im Parallel Sysplex zu diesem

Zweck einen *Sysplex Distributor*, kurz *SD* ([EH+07]). Der Sysplex Distributor repräsentiert den Cluster über eine IP-Adresse nach außen und leitet ankommende Requests über Auswertung der PIs an andere Systeme im Sysplex weiter.

Sollte die LPAR, welche als SD fungiert, ausfallen, übernimmt ein Backup-Host die Aufgabe und wird neuer SD für den Sysplex. Dabei kann die an den SD gebundene IP-Adresse dynamisch an das neue System gebunden werden. Auf diese Weise wird die Netzwerkschnittstelle als *Single Point of Failure* eliminiert.

Die vollen Möglichkeiten des Clustering auf System z mit z/OS lassen sich hier nur andeuten. Als Ergebnis ergeben sich aber folgende Vorteile für den Betrieb eines WebSphere Application Server Clusters auf System z unter z/OS mit Workload Manager und Parallel Sysplex([Lyo07]):

- Ist ein System überlastet, wird es zugunsten weniger ausgelasteter Systeme umgangen.
- Ausgefallene Systeme werden ignoriert. Andere Systeme übernehmen ihre Aufgaben.
- Das ausgefallene System wird, wenn möglich, wieder hergestellt.
- Sollte der gesamte Parallel Sysplex sein Leistungslimit erreichen, werden, durch ausgefeiltes Workloadmanagement, wichtige Dienste bevorzugt und bleiben verfügbar.

Clustering auf System z mit Linux for IBM System z

An dieser Stelle muss eine dritte Unterscheidung eingeführt werden. Bisher wurde nur WebSphere auf z/OS mit WebSphere Distributed auf allen anderen Betriebssystemen verglichen. Was bisher zu WebSphere Distributed gesagt wurde, gilt auch für WebSphere auf Linux for IBM System z.

Linux for IBM System z, kurz z*Linux*, ist ein Linux-Betriebssystem, das für die System z-Hardware kompiliert wurde. Obwohl es im Cluster nicht mit einem Parallel Sysplex mithalten kann, bietet der Betrieb von Linux auf System z immer noch die Vorteile, die sich aus der Hardware eines System z ergeben.

- Breite Anbindung an DASD über FICON/ESCON-Kanäle und eigenen Prozessoren (*System Assist Processor*) zur Steuerung des Datenflusses.
- Kryptographie-Unterstützung in Hardware.
- Virtualisierung unterstützt durch die Hardware.
- Redundanz auf vielen Ebenen zur Minimierung von Ausfallzeiten (doppelte Stromversorgung, Prozessoren, die nur bei Ausfall anderer Prozessoren aktiv werden, ...).
- SMP-Architektur mit bis zu 54 Prozessoren auf einem System.
- Volle Ausnutzung der zur Verfügung stehenden Ressourcen durch Virtualisierung.

zLinux kann nativ auf der Hardware ausgeführt werden, wird aber meistens auf einer virtuellen Maschine ausgeführt. Durch Virtualisierung können viele Systeme parallel auf ein und derselben Hardware laufen. Die virtuelle Maschine übernimmt hierbei die Verwaltung der Hardware für die Gastsysteme und stellt ihnen sämtliche Ressourcen über Schnittstellen zur Verfügung. Das

virtualisierte Gastsystem ist dabei nicht nicht in der Lage zu erkennen, dass es nicht direkt mit der Hardware arbeitet.

Eine Möglichkeit auf System z zu virtualisieren ist z/VM. z/VM besteht aus einem *Control Program*, das die Schnittstelle von Hardware zu Gastsystem ist, und aus einem *Conversational Monitor System*, das die Schnittstelle von Gastsystem zur virtuellen Maschine ist. Die Virtualisierung über z/VM entspricht der Virtualisierung in Software, wenn auch eng verzahnt mit der Hardware durch eigens von z/VM verwendete Prozessorinstruktionen. Die Speicherveraltung wird dabei vom Host-System übernommen, was einen Overhead erzeugt. Dieser Overhead entsteht zusätzliche zum Overhead, der durch die ständigen Kontextwechsel zwischen z/VM und darauf laufenden Systemen entsteht. CPUs und I/O-Kanäle werden von der virtuellen Maschine für das Gast-System simuliert.

Auf einer System z kann auch hardwarenaher virtualisiert werden. Dazu wird der *Processor Ressource/System Manager*, kurz *PR/SM*, verwendet. Der PR/SM teilt hierbei eine Teilmenge der zur Verfügung stehenden Hardware einer logischen Partition, kurz *LPAR*, zu und fungiert als Hypervisor (Überwacher) der untergebenen Systeme. Der Speicherzugriff wird, im Gegensatz zur Verwendung von z/VM, nicht virtualisiert. Jede LPAR hat einen fest zugewiesenen, zusammenhängenden Bereich im Speicher. Das spart den Overhead an Speicherumsetzung im Gegensatz zu z/VM ein. CPUs und I/O-Kanäle können dediziert einer LPAR zugewiesen oder von mehreren LPARs geteilt verwendet werden. In modernen System z gibt es keinen so genannten *Basic Mode* mehr, d.h. die ablaufenden Betriebssysteme werden immer über PR/SM virtualisiert ([EOO06]). Der Performanceverlust durch die Virtualisierung liegt hierbei im einstelligen Prozentbereich.

Virtualisierung dient der Konsolidierung von Servern. Durch die Konsolidierung lässt sich die Hardware effizienter ausnützen und Softwarekosten lassen sich senken. Anstelle vieler Lizenzen für viele Maschinen benötigt man nur noch wenige Lizenzen für eine große Maschine. Ein System z ist in der Lage über 24h hinweg nahezu 100% Ressourcenauslastung zu erreichen. Vergleichbare Datenzentren auf Windows- oder Unix-Basis erreichen oft weniger als 20% Auslastung über einen Tag hinweg ([BED+04]).

z/VM und PR/SM lassen sich kombinieren. Die maximale Zahl der LPARs variiert zwischen 30 und 60, je nach Hardwareausstattung. Die Zahl der Instanzen auf einer z/VM ist durch die Leistung des Systems beschränkt. Experimentell wurden schon rund 96000 Linux-Betriebssysteme auf einem System z gebootet ([vB07]). Das Laufzeitverhalten dürfte hierbei aber, durch das notwendige Paging, sehr schlecht sein.

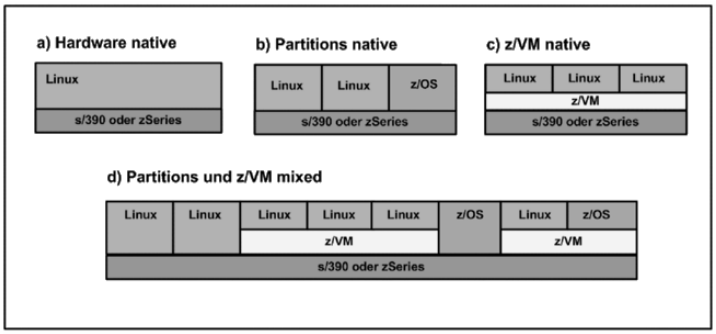


Abbildung 4.11: Kombinationsmöglichkeiten von z/VMs und LPARs ([Spr06]).

Abbildung 4.11 fasst die verschiedenen Betriebsarten zusammen.

a) Nativer Betrieb des Betriebssystems auf der Hardware. Ist auf der neuesten System z nicht mehr möglich. Hier wird immer mindestens in LPARs virtualisiert.

b) Virtualisierung in LPARs über PR/SM.

c) Virtualisierung über z/VM.

d) Kombination aus LPARs, und z/VMs auf den LPARs.

Mittels Virtualisierung lassen sich viele zLinux-Instanzen auf derselben Hardware ausführen. Man kann dies zum Beispiel nutzen, wenn man mehrere WebSphere-Instanzen betreiben möchte, die zu unterschiedlichen Zeiten ihre Lastspitzen erreichen und dabei auf Betriebssystemebene voneinander isoliert sein sollen. Man darf keine Leistungssteigerung erwarten, indem man einfach nur die Zahl der Instanzen erhöht, wenn die Maschine mit weniger Instanzen schon ausgelastet ist. Man kann als weiteres Beispiel auch eine zLinux-Instanz für die Entwicklung, weitere für Tests und wieder weitere für den Betrieb bereitstellen, um so Isolation zwischen den einzelnen Bereichen zu erhalten.

Man kann, über dieselbe Hardware hinweg, mit Partitionen oder virtuellen Maschinen einen horizontalen Cluster aufbauen. Die Netzwerkkommunikation zwischen den einzelnen zLinux-Instanzen kann dabei so konfiguriert werden, dass niemals eine physikalische Netzwerkverbindung benutzt werden muss. z/VM emuliert die Hardwareschnittstellen zwischen den einzelnen Gastsystemen, im speziellen die Netzwerkverbindungen.

Kommt es zur Netzwerkkommunikation zwischen zwei Gastsystemen derselben z/VM-Instanz, erkennt z/VM, dass es Quelle und Ziel der Kommunikation in einem virtuellen *GuestLan* verwaltet und beschleunigt den Datenaustausch dadurch, dass die Daten lediglich im Speicher intern verschoben werden, ohne dass Bits über eine Leitung fließen. Das erhöht auch die Sicherheit der Kommunikation, da die Daten die Maschine nicht verlassen ([BED+04]).

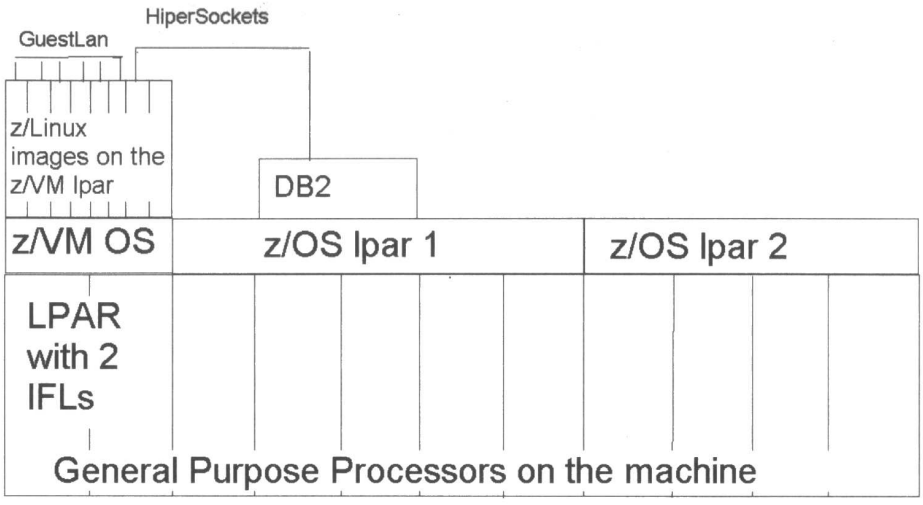


Abbildung 4.12: Virtualisierter Betrieb von Linux on IBM System z ([BED+04]).

Auf gleiche Weise lassen sich partitionenübergreifend sogenannte *Hipersockets* verwenden, um die Kommunikation zwischen zLinux-Gastsystemen und z/OS-Systemen zu beschleunigen, da auch hierbei die physikalische Verbindung umgangen wird (vgl. Abbildung 4.12). Das in dieser Arbeit verwendete Testszenario benutzt zum Beispiel Hipersockets für die Kommunikation zwischen dem WebSphere Application Server in zLinux und der DB2-Datenbank in der z/OS-LPAR. Mehr dazu im nächsten Kapitel.

In Abbildung 4.12 taucht der Begriff *IFL* auf. IFL steht für *Integrated Facility for Linux* und bezeichnet eine sogenannte *Specialty Engine*. Specialty Engines sind eine Eigenheit von System z. Technisch betrachtet ist eine IFL ein *General Purpose Processor* oder auch *General Purpose CPU*, kurz *GPU*, in dem einige z/OS-spezifische Instruktionen deaktiviert wurden ([EOO06]). IFLs sind nur für den Betrieb von Linux auf System z gedacht und können nicht anders verwendet werden. Sie sind in der Anschaffung und im Betrieb günstiger als eine GPU und wurden von IBM eingeführt, um den Betrieb von Linux auf System z betriebswirtschaftlich attraktiv zu machen.

Es gibt auf System z noch weitere Specialty Engines für Java- und DB2-Workload, auf die später noch eingegangen wird.

Die Möglichkeit viele zLinux-Instanzen virtuell auf einer z/VM zu betreiben ist ein großer Vorteil von zLinux auf System z. Ein Problem dabei kann aber der benötigte Speicherplatz auf Festplatte werden. 50 oder mehr zLinux-Instanzen auf einem System z sind keine Seltenheit. Läuft auf jeder der Instanzen ein bestimmte Softwarebestand, wie z.B. ein Webserver, ein WebSphere-Server oder auch eine Datenbank, benötigt eine Instanz bis zu 2GB auf den DASD. Der Platz ist dabei vielleicht nicht einmal das Hauptproblem, sondern die Geschwindigkeitseinbußen, die sich aus dem parallelen Zugriff auf die DASD ergeben können.

Es ist gleichzeitig so, dass ca. 85% dieser Daten nicht exklusiv dieser einen Instanz zur Verfügung stehen müssen. Es muss auf diesen Datenanteil nicht einmal geschrieben werden, d.h. es findet nur lesender Zugriff statt. Mit diesem Hintergrund ist es naheliegend, dass solche Anteile in einer von z/VM verwalteten Minidisk allen zLinux-Instanzen zur Verfügung stehen. Das vermindert einerseits den beanspruchten Platz und andererseits erhöht es die Geschwindigkeit des Zugriffs, da Caching-Mechanismen öfter greifen können ([BED+04]).

zLinux stellt, dank der ausgefeilten Virtualisierung, eine Alternative zu z/OS für den Betrieb von WebSphere auf System z dar.

4.2.4 Fazit: Clustering

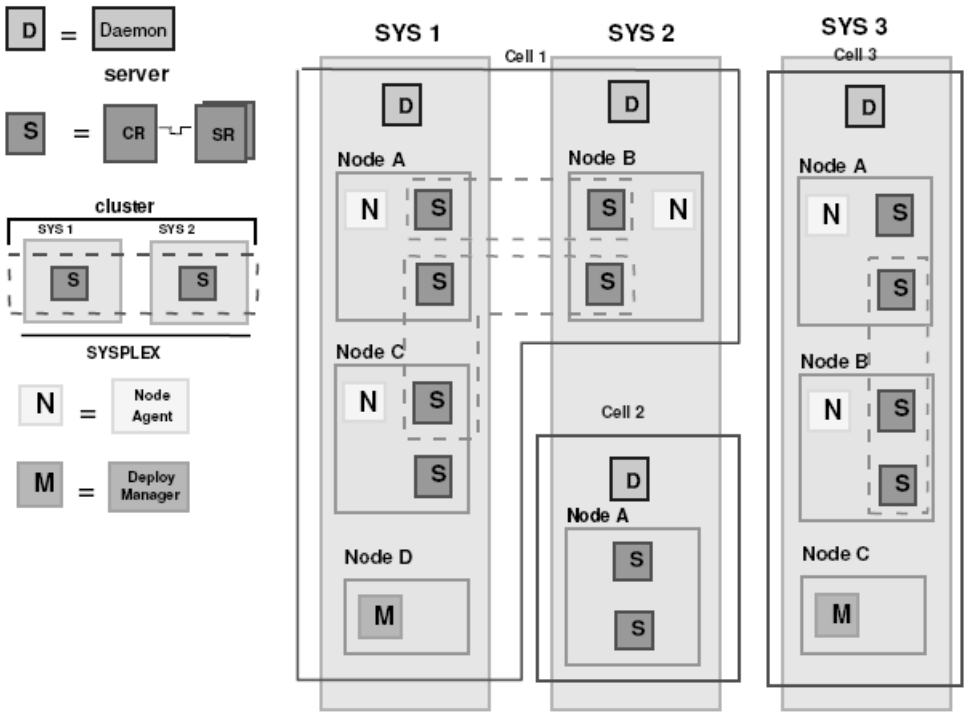


Abbildung 4.13: Clustering-Konzept für den WebSphere Application Server ([IBM08b])

Abbildung 4.13 fasst die bis hier vorgestellten Konzepte noch einmal zusammen. Das Bild zeigt drei Zellen, wobei Zelle 1 sich über zwei Systeme erstreckt und aus drei Nodes mit zwei Clustern besteht. Sollte SYS 1 ausfallen, sind die Anwendungen in den beiden Servern in Node B immer noch erreichbar. Lediglich der Server in Node C, der zu keinem Cluster gehört, wäre nicht mehr erreichbar.

Zelle 3 nimmt exklusiv SYS 3 in Beschlag. Zelle 2 ist ein Einzelserver, der noch nicht einer anderen Zelle hinzugefügt wurde. Hier sieht man auch, dass es im Falle von z/OS als Betriebssystem einen Daemon pro Zelle und LPAR gibt. Denkt man sich die Aufteilung in Control und Servant Region für einen Server, sowie auch die Bezeichnung Sysplex weg, so könnten SYS 1 - SYS 3 beliebige Rechnersysteme auf Windows-Basis, System z-Mainframes oder auch LPARs innerhalb einer System z sein.

4.3 Zusammenfassung

Im vorausgehenden Kapitel wurden die wichtigsten technischen Unterschiede erläutert, die sich für den WebSphere Application Server auf z/OS verglichen mit anderen Systemen ergeben.

Der gravierendste Unterschied ist sicherlich das veränderte Prozessmodell, das sich aus der Nutzung des Workloadmanagers auf z/OS ergibt. Durch das ausgefeilte Workloadmanagement lassen sich ankommende Requests feingranular klassifizieren und bearbeiten.

Für den verteilten Betrieb eines WebSphere Application Server bieten System z und z/OS durch Clustering unter Verwendung spezieller Hardware und guter Virtualisierungs- und Partitionierungsmöglichkeiten Vorteile, was die Ausnutzung zur Verfügung stehender Ressourcen und die Ausfallsicherheit betrifft.

Im folgenden Kapitel werden die für die Tests verwendete Infrastruktur und die verwendeten Werkzeuge vorgestellt.

5 Die Architektur der Testumgebung

Ziel der Arbeit ist die Bewertung zweier möglicher Konfigurationen des Betriebs eines WebSphere Application Servers. Verglichen wird dabei eine zentralisierte Installation mit einer verteilten Installation.

5.1 Hard- und Softwareumgebung

Das Testszenario umfasst zwei LPARs auf einer zSeries an der Universität Leipzig. In einer LPAR läuft z/OS V1.8 als Betriebssystem, auf einer weiteren LPAR läuft z/VM. In der z/VM befinden sich zwei zLinux-Instanzen (SuSE Enterprise Linux Server V9R3). Eine zLinux-Instanz ist nur für das Routing der Anfragen aus dem Internet zuständig. Die Verbindung zur zSeries über das Internet wird über ein *Virtual Private Network*, kurz *VPN*, hergestellt.

Die Vernetzung der beiden zLinux-Instanzen findet über ein GuestLAN der z/VM statt. Die Netzwerkkommunikation zur z/OS-Partition läuft über Hipersockets des PR/SM.

Beiden LPARs ist dediziert eine GPU zugeordnet. In späteren Testläufen wird der z/OS-LPAR noch eine zweite GPU zugewiesen, um Gleichheit bei den zur Verfügung stehenden Ressourcen in beiden Szenarien zu erhalten. LPAR 1 besitzt 4GB Hauptspeicher und LPAR 2 2GB Hauptspeicher.

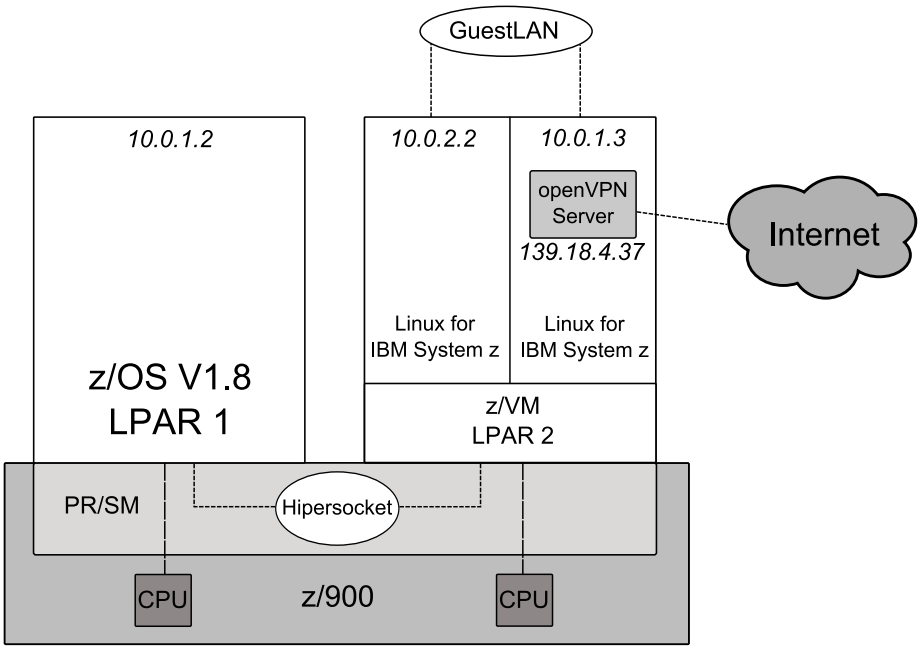


Abbildung 5.1: Überblick über die verwendete LPAR-Architektur.

Das integrierte Szenario

Im integrierten Szenario befinden sich die Datenbank (IBM DB2 for z/OS V8) und der Anwendungsserver (WebSphere Application Server for z/OS V6.1) gemeinsam in der z/OS-Instanz der LPAR 1. Die Kommunikation zwischen Anwendung und Datenbank geschieht über einen JDBC-Treiber Typ 2.

Das verteilte Szenario

Im verteilten Szenario befindet sich die Datenbank ebenfalls in LPAR 1 unter z/OS (IBM DB2 for z/OS V8). Der Anwendungsserver (WebSphere Application Server Network Deployment V6.1) befindet sich als Stand-Alone-Installation in der zLinux-Instanz mit IP 10.0.2.2 in LPAR 2. Die Kommunikation zwischen Anwendung und Datenbank geschieht über einen JDBC-Treiber Typ 4.

5.1.1 Der JDBC-Treiber zur Anbindung der Datenbank

JDBC wurde in seinen Grundzügen bereits auf Seite 20 als Teil der Java EE-Spezifikation vorgestellt. JDBC ermöglicht drei Dinge ([JDB08]):

- Aus einer Java-Anwendung heraus eine Verbindung zu einer relationalen Datenbank oder einer anderen, tabellenbasierten Datenquelle herstellen.
- Versenden von SQL-Anfragen über diese Verbindung.
- Verarbeiten der von der Datenbank zurückgesendeten Daten in eine anwendungsverträgliche Darstellung.

Die JDBC-API ist in zwei Programmierschnittstellen unterteilt:

- Eine JDBC-API für den Anwendungsentwickler zur Anforderung von Verbindungen und zum Absenden von SQL-Queries.
- Eine JDBC-Treiber-API für die Erstellung von Datenbanktreibern durch die Datenbank-Hersteller.

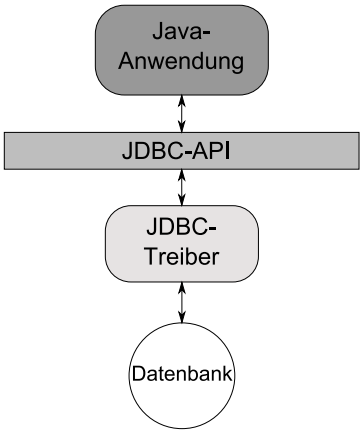


Abbildung 5.2: JDBC-Überblick

Die JDBC-API ermöglicht es dem Anwendungsentwickler unabhängig von der verwendeten Datenbank auf relationale Datenbanken zuzugreifen. Die tatsächliche Anfrage und Verbindungsherstellung geschieht durch den verwendeten Treiber, der vom Datenbankhersteller geliefert wird. Durch das zweischichtige Modell wird hier wieder das Java-Prinzip „*Write once, run anywhere!*“ umgesetzt.

Es gibt derzeit vier Typen von JDBC-Treibern. Typ 1 und Typ 3 sind für diese Arbeit uninteressant und werden von DB2 auf z/OS nicht unterstützt ([SB+05]).

JDBC-Treiber Typ 2

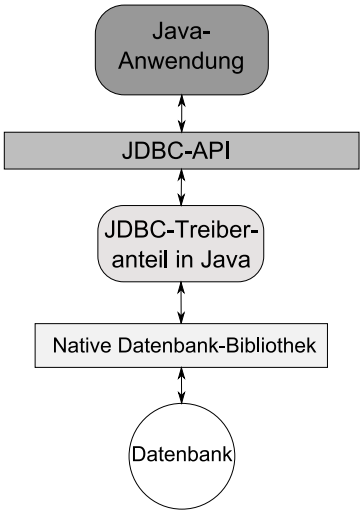


Abbildung 5.3: JDBC-Treiber Typ 2

Der JDBC-Treiber Typ 2 besteht zu einem Teil aus Java-Code und zum anderen Teil aus nativem Datenbank-Code, der als Bibliothek vom Datenbankhersteller bereitgestellt wird. Der Javaanteil übersetzt die Anfrage in native Bibliotheksaufrufe. Durch den nativen Code wird die Portabilität auf die verwendete Plattform eingeschränkt. Dieser Treiber bietet aber die beste Performance für den Fall, dass sich die Datenbank und die darauf zugreifende Anwendung gemeinsam in einem System befinden ([KC+06]). Der Zugriff findet hier komplett über den Speicher und lokale Konnektoren statt und es entsteht kein Overhead durch ein verwendetes Netzwerk. Ganz im Unterschied zu folgendem Treiber.

JDBC-Treiber Typ 4

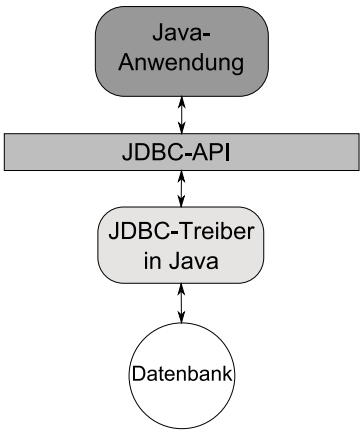


Abbildung 5.4: JDBC-Treiber Typ 4

Der JDBC-Treiber Typ 4 ist komplett in Java geschrieben und somit plattformunabhängig. Der Treiber implementiert ein Datenbankprotokoll für eine spezifische Datenbank. Im Falle von DB2 wird das *DRDA*-Protokoll (*Distributed Relational Database Architecture*) verwendet. DRDA ist ein universelles Protokoll für den Zugriff auf verteilte Daten. Im OSI-Schichtenmodell ist es in den Schichten 5 bis 7 einzuordnen. DRDA bietet keine API an, sondern spezifiziert lediglich die Architektur des verteilten Datenzugriffs und ist keine Implementierung.

Bei entferntem Zugriff über einen Typ 4-Treiber hat man immer einen Overhead für die Netzwerkkommunikation zu erwarten. Es findet auf beiden Seiten der Kommunikation Datenmarshalling statt und der TCP/IP-Stack wird durchlaufen.

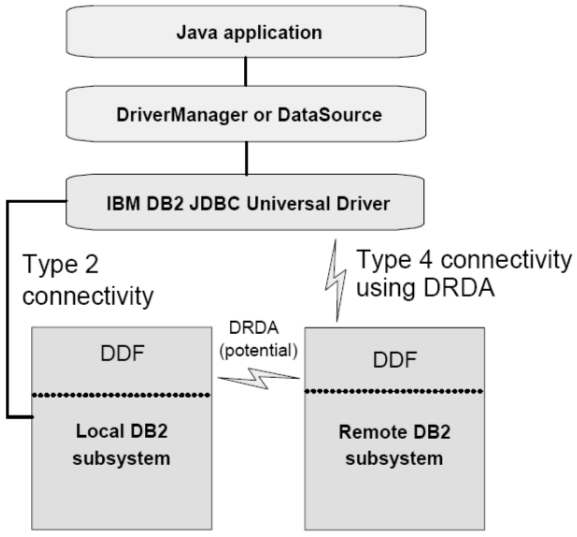


Abbildung 5.5: JDBC-Treiber Typ 2 und 4 ([SB+05])

Für DB2 liefert IBM einen optional installierbaren *DB2 Universal Driver for Java Common Connectivity* mit. Dieses Paket enthält JDBC-Treiber des Typs 2 und 4 und einen SQLJ-Treiber. Letzteres wird in dieser Arbeit nicht verwendet und auch nicht behandelt.

Um aus einer Java EE-Anwendung im WebSphere Application Server auf eine DB2 UDB zugreifen zu können, wird im WAS eine sogenannte *Data Source* (Datenquelle) über den JNDI-Service registriert. Die Data Source ist ein Objekt, das im Anwendungsserver die an es gebundene Datenquelle, in diesem Fall die DB2 UDB, repräsentiert (vgl. Abbildung 5.5). In den Properties des Objekts stehen die Eigenschaften der Datenbank, wie z.B. Server-IP, Portnummer, usw.

Durch die Registrierung der Datenbank über den JNDI-Service wird die Datenbank der Anwendung gegenüber austauschbar. In der Anwendung müssen keine hartkodierten Informationen über die verwendete Datenbank hinterlegt werden. Der Zugriff auf eine Datenbank aus der Anwendung heraus läuft in 3 Schritten ab ([uFA+06]):

- Die Anwendung referenziert über den JNDI-Service ein DataSource-Objekt.
- Nachdem das Objekt instanziiert ist, wird durch den Aufruf der `getConnection()`-Methode, welche im DataSource-Objekt implementiert ist, ein Verbindungs-Objekt für die Datenbank angefordert. Diese Verbindungen werden in einem Connection-Pool vorgehalten und nicht immer neu erzeugt.
- Über die so hergestellte Verbindung werden Anfragen und Updates an die Datenbank gesendet.

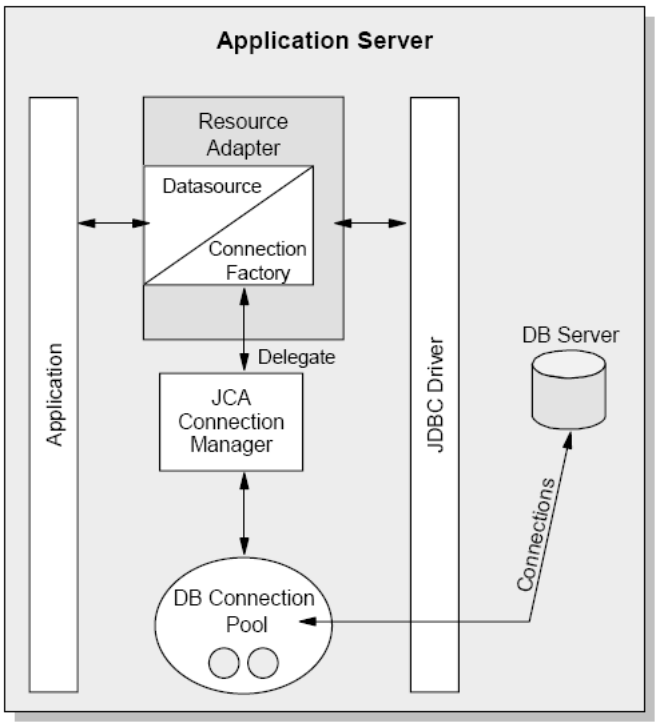


Abbildung 5.6: Herstellung einer Verbindung zur Datenbank ([uFA+06]).

DB2 auf z/OS

Eine DB2-Instanz auf z/OS besteht aus mehreren Adressräumen. Adressraumbezeichnungen für DB2 haben die Form *ssnmxxxx*, wobei *ssnm* der Bezeichner des DB2-Subsystems ist, z.B. D931, und *xxxx* den jeweiligen Adressraum bezeichnet ([SB+05]).

- *ssnmMSTR:* Der *Master*-Adressraum ist verantwortlich für Speicherverwaltung, Prozedurinitialisierung, Logging, Recovery, etc.
- *ssnmDBM1:* Hier werden die meisten DB2-Operationen ausgeführt, z.B. Stored Procedures, Daten- und Buffermanagement, etc.
- *ssnmIRLM:* Der *Internal Resource Lock Manager* ist für die Integrität der Daten in der Datenbank verantwortlich.
- *ssnmDIST:* In diesem Adressraum läuft die *Distributed Data Facility*, kurz *DDF*. Sie ist der Kommunikationsendpunkt für entfernte Anfragen an die Datenbank.

DDF wird für die Messungen im verteilten Szenario eine große Rolle spielen. Mit diesem Adressraum stellen Anwendungen eine Verbindung über einen JDBC-Treiber Typ 4 her.

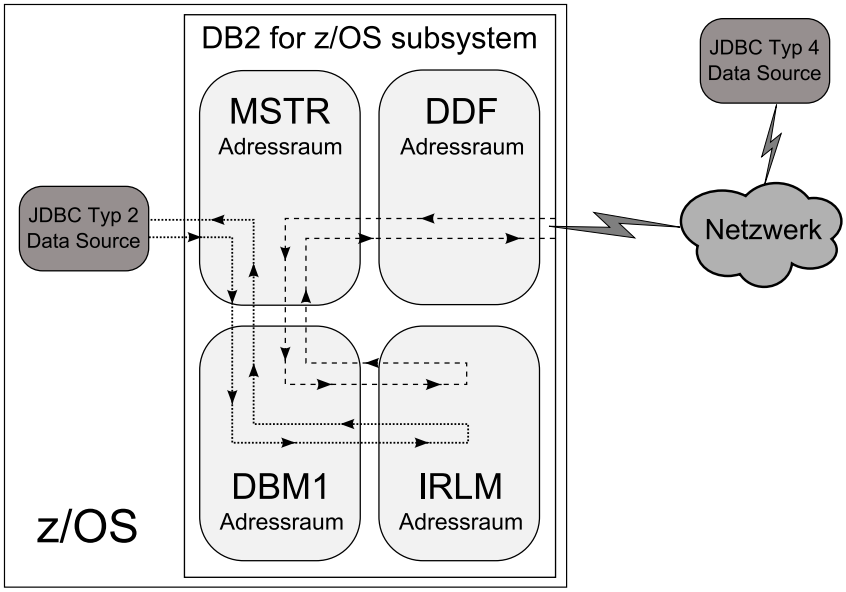


Abbildung 5.7: Fluss einer DB2-Verbindung durch die Adressräume des DB2-Subsystems auf z/OS.

Eine JDBC Type 2-Verbindung greift auf die Datenbank direkt über den MSTR-Adressraum zu. Eine JDBC Type 4-Verbindung läuft zuerst über den DDF-Adressraum und erreicht danach erst den MSTR-Adressraum. Danach läuft die Anfrage für beide Szenarien gleich ab ([SB+05]).

- Der MSTR-Adressraum validiert die Anfrage und sendet gegebenenfalls erste Antwortnachrichten an die Ursprungsanwendung.

- Der Datenfluss geht weiter in den DBM1-Adressraum, wo die Anfrage im eigentlichen Sinne bearbeitet wird.
- Der IRLM-Adressraum nimmt das benötigte Locking der Daten vor, die bearbeitet werden.
- Nachdem die Datenintegrität gesichert ist, werden die angeforderten Daten oder eine Bestätigung der vorgenommenen Aktionen an den Requester zurück gesendet.

Der grundlegende Unterschied zwischen den verwendeten Treibern ist die Verwendung des DDF-Adressraumes bei entfernten Anfragen. Dieser nimmt DRDA-konforme Anfragen an und wandelt sie in Anfragen der spezifischen Datenbank um. Diesen Schritt erledigt beim JDBC-Treiber Typ 2 bereits der Treiber selbst, der die Anfragen direkt in nativen Datenbank-Code umwandelt.

5.1.2 Zusammenfassung

Es gibt zwei Konfigurationen des WebSphere Application Server, die in dieser Arbeit verglichen werden. Die folgenden Abbildungen stellen die beiden Konfigurationen noch einmal zusammengefasst dar:

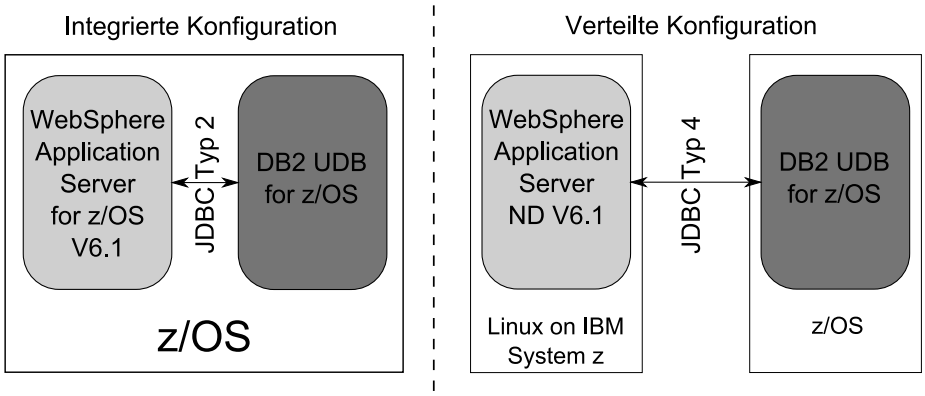


Abbildung 5.8: Die beiden Konfigurationen für die Performanceuntersuchung.

5.2 Die Benchmarkanwendung: IBM Trade V6.1

TradeV6.1 ist eine Benchmark-Anwendung für den WebSphere Application Server von IBM. Sie erfüllt die J2EE-Spezifikation 1.4. TradeV6.1 ist einer einfachen Aktienhandelsplattform nachempfunden und wurde speziell als End-To-End-Benchmark für den WebSphere Application Server entwickelt. Abbildung 5.9 zeigt einen Überblick über die Topologie der Trade6-Anwendung.

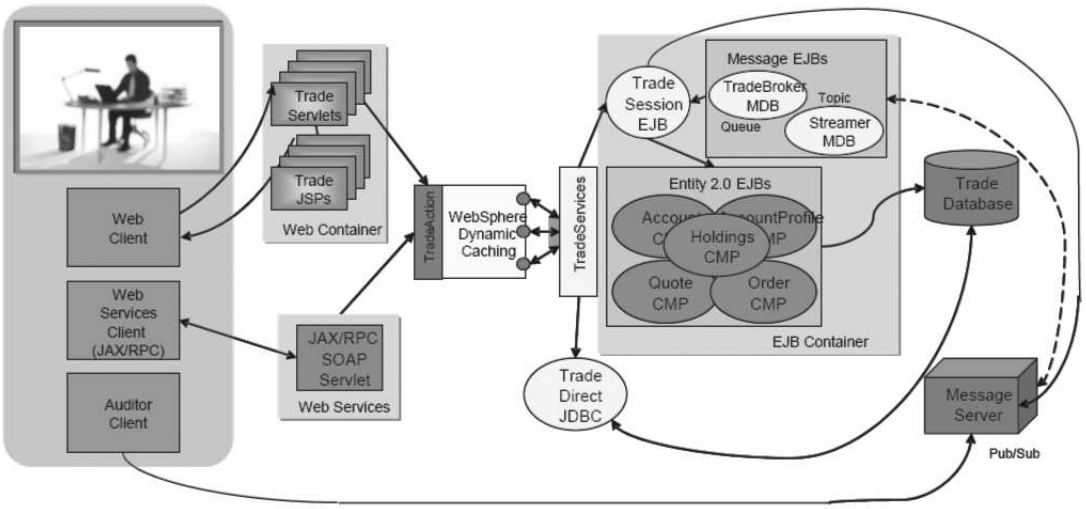


Abbildung 5.9: Topologie des TradeV6.1-Benchmarks

TradeV6.1 bietet ein zentrales Service-Interface für alle Clientarten. Für die Performanceuntersuchungen in dieser Arbeit wird ein Webclient benutzt. Das Interface wird durch ein Stateless Session Bean implementiert.

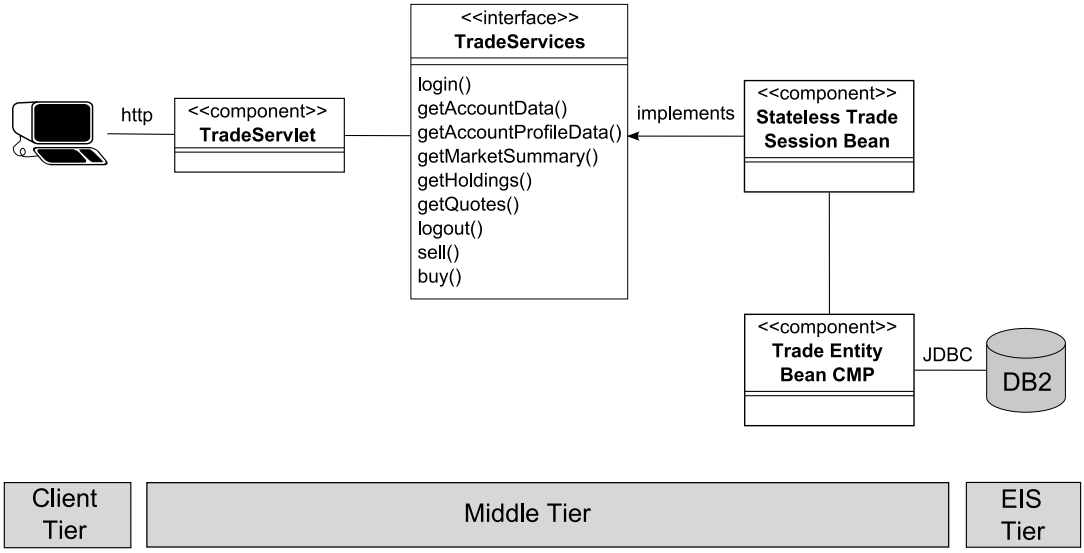


Abbildung 5.10: TradeV6.1 Anwendungsarchitektur (vereinfacht!).

Jede High-Level-Benutzeroperation, in diesem Fall ein HTTP-Request, löst eine Reihe von Methodenaufrufen im SessionBean und eine Reihe von Operationen auf der Datenbank aus. Die *Trade Runtime and Database Usage Characteristics* ([IBM06]) gibt Auskunft darüber, welcher Art die Operationen auf der Datenbank sind. Um herauszufinden, welche Methodenaufrufe eine High-Level-Operation auslöst, lassen sich *Request Metrics* verwenden.

Request Metrics sind ein im WebSphere Application Server integriertes Werkzeug, um den Fluss einer Transaktion im Anwendungsserver zu verfolgen. In der `SystemOut.log`-Datei werden je nach eingestelltem Tracelevel die Methodenaufrufe zusammen mit der Dauer des Aufrufs protokolliert. Je nach Tracelevel kann es auch zu großem Overhead durch das Tracing kommen. In Produktivsystemen sollten Request Metrics daher nur mit Bedacht erfasst werden. Auf der CD findet sich ein Logauszug unter `requestmetrics/zLinuxSystemOut.log`. Dieser wurde bei High-Level-Aufrufen, die auch während der Messungen verwendet werden, erfasst. Als Tracelevel war *Performance_debug* eingestellt.

Aufruf der Loginpage

URI-Aufruf: /trade/app

Eine statische HTML-Seite mit einem Login-Formular wird vom WebContainer zurück geliefert.

Login

URI-Aufruf: /trade/app **Action:** login **Parameter:** uid, passwd

Auf dem Client wird ein Cookie angelegt, das eine Session-ID der im Server erzeugten Session enthält, um den Client für den weiteren Verlauf der Sitzung identifizierbar zu machen.

Zu Anschauungszwecken werden hier einmalig die mitgeloggten Methodenaufrufe aufgeführt. Diese sind nicht als JVM-Trace zu verstehen. Tatsächlich fand eine große Menge mehr Methodenaufrufe und Objektinteraktion statt, die über Request Metrics mit dem Tracelevel *Debug* oder direkt über ein Tracing der JVMs mitzuloggen wären.

```
com.ibm.websphere.samples.trade.ejb.TradeBean.login()
    javax.resource.spi.ManagedConnectionFactory.matchManagedConnections()
    javax.resource.spi.ManagedConnection.getConnection()
    javax.resource.spi.LocalTransaction.begin()
    java.sql.PreparedStatement.executeQuery()
    javax.resource.spi.ManagedConnection.getConnection()
    java.sql.PreparedStatement.executeQuery()
    javax.resource.spi.ManagedConnection.getConnection()
    java.sql.PreparedStatement.executeUpdate()
        java.sql.Connection.commit()
    javax.resource.spi.LocalTransaction.commit()
    javax.resource.spi.ManagedConnection.cleanup()
com.ibm.websphere.samples.trade.ejb.TradeBean.getAccountData()
    javax.resource.spi.ManagedConnectionFactory.matchManagedConnections()
    javax.resource.spi.ManagedConnection.getConnection()
    javax.resource.spi.LocalTransaction.begin()
    java.sql.PreparedStatement.executeQuery()
    javax.resource.spi.ManagedConnection.getConnection()
    java.sql.PreparedStatement.executeQuery()
        java.sql.Connection.commit()
    javax.resource.spi.LocalTransaction.commit()
    javax.resource.spi.ManagedConnection.cleanup()
com.ibm.websphere.samples.trade.ejb.TradeBean.getHoldings()
    javax.resource.spi.ManagedConnectionFactory.matchManagedConnections()
    javax.resource.spi.ManagedConnection.getConnection()
    javax.resource.spi.LocalTransaction.begin()
```

```
    java.sql.PreparedStatement.executeQuery()
    javax.resource.spi.ManagedConnection.getConnection()
    java.sql.PreparedStatement.executeQuery()
    javax.resource.spi.ManagedConnection.getConnection()
    java.sql.PreparedStatement.executeQuery()
    javax.resource.spi.ManagedConnection.getConnection()
    java.sql.PreparedStatement.executeQuery()
        java.sql.Connection.commit()
    javax.resource.spi.LocalTransaction.commit()
    javax.resource.spi.ManagedConnection.cleanup()
com.ibm.websphere.samples.trade.ejb.TradeBean.getMarketSummary()
    javax.resource.spi.ManagedConnectionFactory.matchManagedConnections()
    javax.resource.spi.ManagedConnection.getConnection()
    javax.resource.spi.LocalTransaction.begin()
    java.sql.PreparedStatement.executeQuery()
        java.sql.Connection.commit()
    javax.resource.spi.LocalTransaction.commit()
    javax.resource.spi.ManagedConnection.cleanup()
```

In einem ersten Aufruf der Methode `login()` wird über eine JDBC-Verbindung der Login abgehandelt. Es werden zwei Datenbankanfragen gestellt und ein Update der Accountdaten durchgeführt, z.B. Anzahl der Logins. Durch Aufruf der Methoden `getAccountData()`, `getHoldings()` und `getMarketSummary()` werden danach die für die Home-Seite notwendigen Daten aus der Datenbank geholt. Dazu sind insgesamt sieben SQL-Anfragen notwendig, die über drei JDBC-Verbindungen abgehandelt werden.

Check Account

URI-Aufruf: /trade/app **Action:** account

Hier werden die Methoden `GetAccountData` und `getAccountProfileData` des TradeBean-Objekts verwendet, um die benötigten Daten aus der Datenbank abzufragen. Dazu werden zwei JDBC-Verbindungen aufgebaut und vier Anfragen gestellt.

Switch to home

URI-Aufruf: /trade/app **Action:** account

Ein Wechsel zur Home-Webseite entspricht exakt der angezeigten Seite nach dem Login und wird, abgesehen vom Loginvorgang und dem Update der Accountdaten, nach demselben Ablauf erstellt.

Get quotes

URI-Aufruf: /trade/app **Action:** quotes **Parameter:** symbols

Das Abrufen fiktiver Börsennotierungen ist eine der länger dauernden Aktionen in Trade und in seinem Umfang und seiner Dauer abhängig von der Anzahl der angeforderten Notierungen. In den Messungen werden elf zufällig ausgewählte Aktien abgerufen. Für jede der Aktien wird eine Anfrage an die Datenbank gesendet, wobei jedes Mal eine neue Verbindung aus dem Verbindungspool angefordert und wieder abgegeben wird.

Buy a stock

URI-Aufruf: /trade/app **Action:** buy **Parameter:** symbol, quantity

In der Methode `buy()` werden über eine JDBC-Verbindung vier Anfragen und vier Updates an die Datenbank gesendet. Für die Neuberechnung des fiktiven Kurses, die einem Kauf folgt, werden in der Methode `updateQuotePriceVolume()` über zwei JDBC-Verbindungen eine Anfrage und zwei Updates auf der Datenbank ausgeführt.

Auf der als nächstes angezeigten Webseite wird, wenn die Transaktion erfolgreich durchgeführt wurde, immer eine Bestätigung der erfolgreichen Durchführung angezeigt.

Switch to portfolio

URI-Aufruf: /trade/app **Action:** portfolio

Hier ist die Anzahl der Datenbankanfragen innerhalb der Methode `getHoldings()` wieder abhängig von der Anzahl der im Besitz befindlichen Aktien des Benutzers. Innerhalb der Methode werden alle Anfragen über eine Verbindung abgehandelt. Im Anschluss daran wird für jede Aktie ein Aufruf der Methode `getQuote` mit jeweils einer Anfrage über eine JDBC-Verbindung durchgeführt. Die Antwortseite setzt sich dann aus dem Ergebnis dieser Anfragen zusammen.

Sell a stock

URI-Aufruf: /trade/app **Action:** sell **Parameter:** holdingID

In der Methode `sell()` werden über eine JDBC-Verbindung sechs SQL-Anfragen und fünf Updates durchgeführt. In der Methode `updateQuotePriceVolume()` wird über eine JDBC-Verbindung eine Datenbankabfrage und über zwei weitere JDBC-Verbindungen zwei Updates der Datenbank durchgeführt. Wie beim Kauf einer Aktie wird als nächstes eine Transaktionseingangsbestätigung und auf der beliebigen nächsten Seite eine Transaktionsbestätigung angezeigt.

Logout

URI-Aufruf: /trade/app **Action:** logout

Bei einem Logout werden über eine JDBC-Verbindung zwei Queries und ein Update an die Datenbank gesendet.

5.2.1 Die Verwendung von MDBs

Die Methoden `sell()` und `buy()` verwenden Message Driven Beans mit queuebasiertem und Publish/Subscribe-Nachrichtenaustausch.

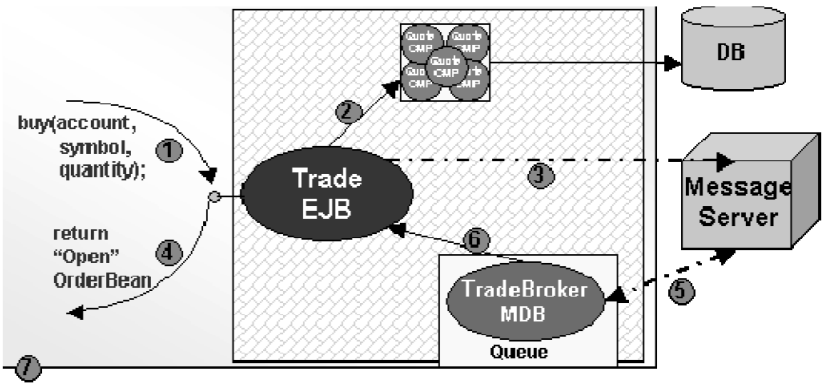


Abbildung 5.11: Ablauf eines Aktienkaufs unter Verwendung von Message Driven Beans.

1. Der Benutzer sendet eine Kaufanfrage ab.
2. Eine neue Aktienbestellung wird in der Datenbank hinterlegt. Diese bleibt auf weiteres „offen“.
3. Die Kaufanfrage wird in der TradeBrokerQueue eingereiht.
4. Der Erhalt und die Erstellung der Kaufanfrage wird dem Benutzer bestätigt
5. Der Nachrichtenserver leitet die Anfrage an den Nachrichtenkonsumenten (TradeBroker-MDB) weiter.
6. Die Anfrage wird vom MDB asynchron bearbeitet.
7. Bei einem der folgenden Requests erhält der Benutzer eine Bestätigung des ausgeführten Kaufs.

Folgende Merkmale qualifizieren Trade V6.1 als Benchmark für die hier vorzunehmende Untersuchung ([KC+06]):

- Unterstützung einer 3-Tier-Architektur.
- Verwendung einer Datenbank zur Datenhaltung.
- Breite Verwendung von Java EE-Komponenten, wie Java Server Pages, Servlets, EJB Session und Entity Beans und Message Driven Beans.

5.3 Der Lastgenerator: Apache jMeter

Für Performanceuntersuchungen soll die Java EE-Anwendung unter möglichst realistischen Bedingungen getestet werden. Zur Erzeugung der dazu notwendigen Anfragenmenge wird ein Lastgenerator oder auch Workloadgenerator verwendet. In dieser Arbeit ist dies die *jMeter*-Anwendung des Apache Jakarta Projekts (http://jakarta.apache.org).

jMeter ist eine Open-Source Java-Anwendung, die ursprünglich für das Testen von Webseiten über das HTTP-Protokoll entwickelt wurde, sich aber seither stetig weiterentwickelt hat. jMeter bietet jetzt jetzt eine breite Unterstützung für das funktionale und leistungstechnische Messen von Webangeboten. jMeter unterstützt statischen und dynamischen Webinhalt (HTTP- und FTP-Server, Servlets, Perl-Skripte, Java-Objekte, Datenbanken) und eine Vielzahl von Protokollen (HTTP, HTTPS, FTP, SOAP, JMS, JNDI und LDAP) ([KC+06]). jMeter läuft auf jeder Plattform, die eine *Java Runtime Environment* ab Version 1.3 unterstützt. Über einen Plugin-Mechanismus kann jMeter um weitere Funktionen erweitert werden.

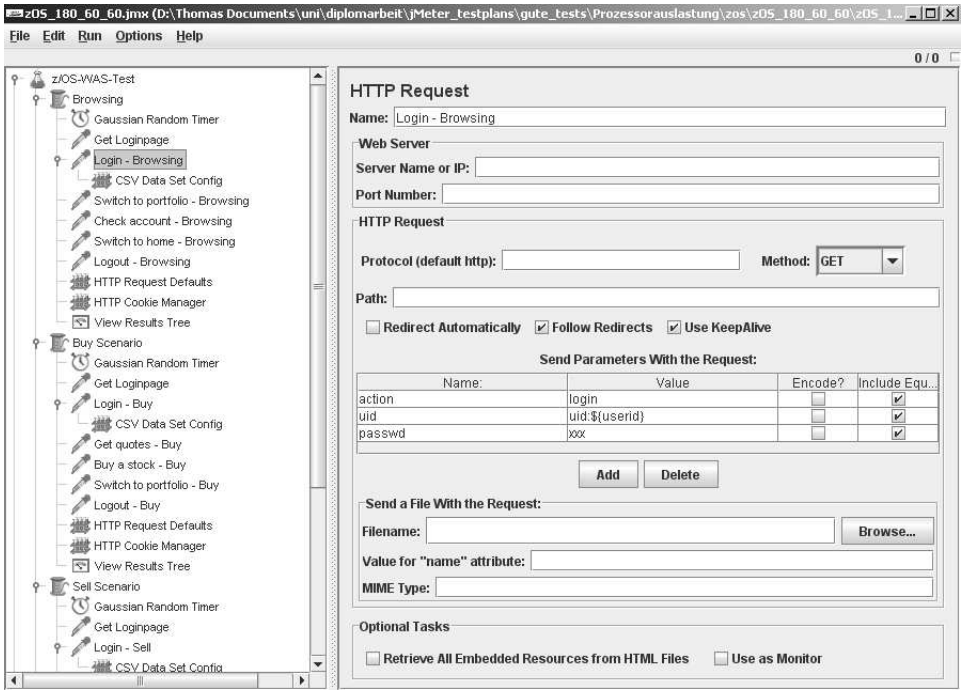


Abbildung 5.12: Screenshot der jMeter-GUI.

Die grafische Oberfläche basiert auf der Java-Swing-API und ist für die Erstellung von Testszenarien notwendig. Der eigentliche Lasttest kann ohne grafische Oberfläche gestartet werden, um die Ressourcen des Testsystems zu schonen. Grundlegendes Konfigurationsmittel sind sogenannte *Configuration Elements*, die zum Beispiel einen HTTP-Request oder JDBC-Request repräsentieren und ausführen.

Neben den Möglichkeiten zur Erzeugung von Lasttests können in jMeter grafische und numerische Auswertungen der Testverläufe vorgenommen werden. In dieser Arbeit wird zum Beispiel ein aggregierter Report benutzt, um die Antwortzeiten und den Durchsatz der Szenarien aus Clientsicht zu vergleichen.

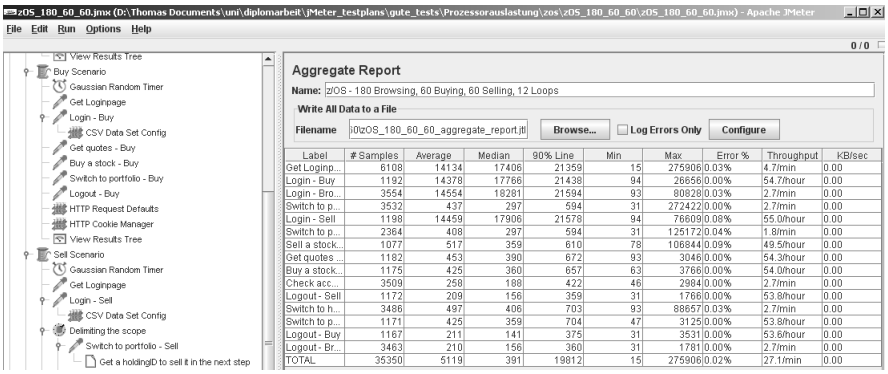


Abbildung 5.13: Aggregierter Report eines Testlaufes in jMeter.

Die Konfiguration von jMeter für die hier vorgenommene Untersuchung ist in Kapitel 6 ab S. 67 im Detail nachzulesen.

5.4 Erfassen der Ressourcenverbräuche

Im Zentrum dieser Arbeit steht ein Vergleich der Prozessorlast, die auf den beiden Systemkonfigurationen durch ankommende Last verursacht wird. Um diese Last zu vergleichen muss sie zuerst erfasst werden. Die beiden Betriebssysteme Linux und z/OS bieten hierzu unterschiedliche Möglichkeiten. Die erfassten Daten der beiden Betriebssysteme sind allerdings häufig nicht direkt vergleichbar, da unterschiedliche Metriken benutzt werden. IBM bietet allerdings für beide Plattformen ein Werkzeug zur Ressourcenüberwachung an, das sogenannte *RMF Performance Monitoring Java Technology Edition*, kurz *RMF-PM*.

RMF-PM ist ein Frontend zur Überwachung von z/OS oder Linux-Systemen (z- oder Intel-basierend). Die angezeigten Systemdaten erhält das Frontend im Falle von z/OS aus einem *RMF Monitor III* und im Falle von Linux aus dem *RMF Linux Data Gatherer*.

5.4.1 RMF auf z/OS

RMF steht für *Resource Measurement Facility* und ist das Werkzeug zur Systemanalyse in z/OS. RMF unterstützt die kurzzeitige Überprüfung des Systemzustands, sowie auch Langzeitüberwachung und Auswertung historischer Daten. RMF erhält seine Daten von drei Monitoren, die z/OS-Ressourcen überwachen ([CK+05]):

- **Monitor I:** Monitor I ist ein Langzeit-Monitor, der zu einer bestimmten Zykluszeit (engl. *Cycle Time*) den Systemzustand erfasst und in bestimmten Intervallen einen konsolidierten Report erstellt. Die Intervallzeit beträgt für gewöhnlich 15 bis 30 Minuten und die Reports werden über Tage/Wochen gesammelt, um dann eine Langzeitanalyse durchzuführen ([CK+05]). Monitor I erfasst dabei alle Hard- und Softwarekomponenten eines Systems, wie zum Beispiel Prozessoren, I/O-Geräte und -Pfade, Speicheraktivitäten, Adressräume, etc.

- **Monitor II:** Monitor II ist ein so genannter *Snapshot-Monitor* und er erfasst den Zustand von Adressräumen und Ressourcen zu einem bestimmten Zeitpunkt (Momentaufnahme).

Er wird benutzt, wenn man einen speziellen Ablauf im System überprüfen und beobachten möchte. Im Unterschied zu Monitor III zeigt Monitor II den Zustand des Systems zum Zeitpunkt der Anfrage, während Monitor III z.B. den Zustand des Systems in den letzten 100 Sekunden darstellt. Monitor II ist vergleichbar mit dem Programm *top* auf Unix- und Linux-Systemen.

- **Monitor III:** Monitor III kann wie Monitor I zur Langzeitanalyse eingesetzt werden. Die Standardeinstellung für die Zykluszeit und die Intervallzeit sind aber eine Sekunde bzw. 100 Sekunden. Monitor III kann somit auch zur Kurzzeitanalyse eingesetzt werden. Die Intervallzeit wurde für die hier gemachten Untersuchungen auf 60s geändert, um mit der Einstellung des RMF Data Gatherers des Linux-Systems kohärent zu sein. Im Gegensatz zu Monitor II zeigt Monitor III auch historische Daten an.

Abbildung 5.14 gibt einen Überblick über die weiteren Komponenten der RMF-Umgebung.

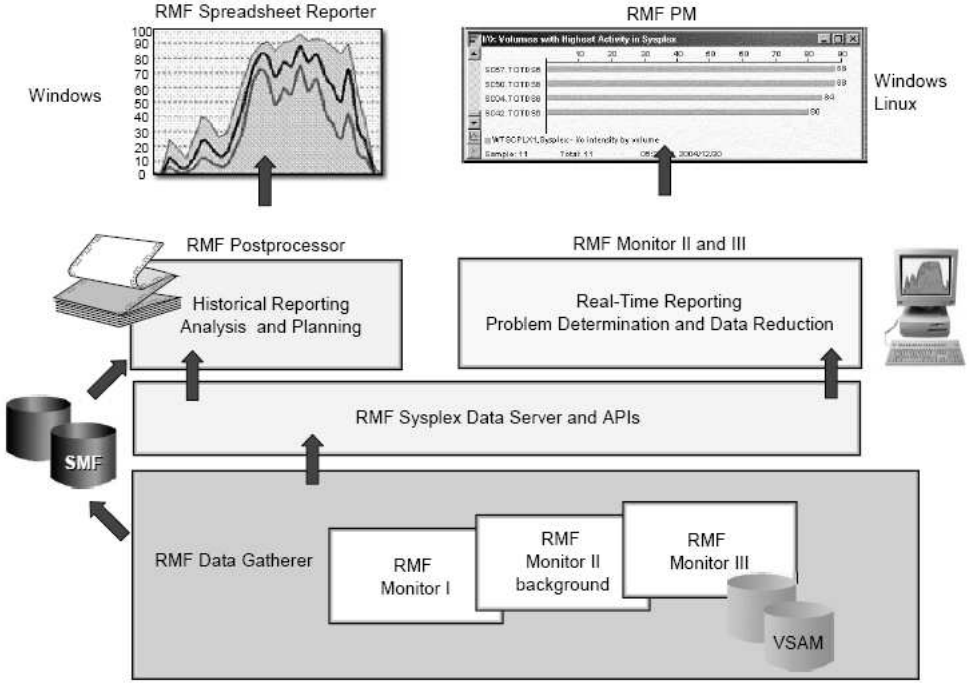


Abbildung 5.14: Überblick über RMF ([CK+05]).

Alle drei Monitore sind Teil der RMF-Gatherer-Instanz, einem eigenen z/OS-Adressraum. Der *RMF Sysplex Data Server* ist nur für einen Parallel Sysplex von Belang. Auf jedem Teilnehmer des Sysplexes läuft eine SDS-Instanz, die mit anderen SDS-Instanzen kommuniziert und z.B. die Intervallzeitpunkte synchronisiert. Weiterhin bietet SDS von jedem System aus einen Einstiegspunkt in die RMF-Überwachung. Mit SDS lässt sich ein Sysplex als Ganzes überwachen und analysieren.

Über die RMF API bietet RMF eine Programmierschnittstelle an, um aus eigenen Programmen heraus auf RMF-Daten zugreifen zu können.

Der Post-Processor fasst RMF-Daten, die in SMF-Data-Sets gespeichert sind, in konsolidierten Reports zusammen. Mit dem Post-Processor kann man aus den gesammelten Daten Reports erstellen lassen, die genau die Informationen über einen gewünschten Zeitraum enthalten, die man für die Analyse benötigt. Die Möglichkeiten des Post-Processors sind mannigfaltig und übersteigen den Umfang dieser Arbeit. Er wird in dieser Arbeit nicht verwendet, da es kein vergleichbares Werkzeug unter Linux gibt. Abbildung 5.15 zeigt einen Workload Activity Report des WLM Workloads *TSO*, der mit Hilfe des Post-Processors erstellt wurde.

```
                                W O R K L O A D   A C T I V I T Y
                                                                                                PAGE  20
     z/OS V1R6              SYSPLEX UTCPLXJ8          DATE 10/14/2004          INTERVAL 30.00.004   MODE = GOAL
                            RPT VERSION V1R5 RMF      TIME 12.30.00

                                      POLICY ACTIVATION DATE/TIME 10/01/2004 14.35.19

REPORT BY: POLICY=WLMPOL01   WORKLOAD=TSO        SERVICE CLASS=TSO        RESOURCE GROUP=*NONE      PERIOD=1 IMPORTANCE=2
                                                 CRITICAL     =NONE

TRANSACTIONS    TRANS.-TIME  HHH.MM.SS.TTT   --DASD I/O--   ---SERVICE----   --SERVICE TIMES--   PAGE-IN RATES    ----STORAGE----
AVG     29.01   ACTUAL             1.054   SSCHRT  94.4   IOC      3017K   TCB          617.4   SINGLE     0.0   AVG     1943.52
MPL     29.00   EXECUTION          1.054   RESP     1.8   CPU     48424K   SRB           25.8   BLOCK      0.0   TOTAL   56355.0
ENDED   47460   QUEUED                0   CONN     1.0   MSO     17420K   RCT           17.9   SHARED     0.0   CENTRAL 56355.0
END/S   26.37   R/S AFFINITY          0   DISC     0.2   SRB      2021K   IIT            2.1   HSP        0.0   EXPAND      0.00
#SWAPS  40073   INELIGIBLE            0   Q+PEND   0.6   TOT     70882K   HST            0.0   HSP MISS   0.0
EXCTD       0   CONVERSION            0   IOSQ     0.0   /SEC    39379   IFA            N/A   EXP SNGL   0.0   SHARED    26.24
AVG ENC  0.00   STD DEV           2.323                                   APPL% CP      36.8   EXP BLK    0.0
REM ENC  0.00                                            ABSRPTN  1358    APPL% IFACP    0.0   EXP SHR    0.0
MS ENC   0.00                                            TRX SERV 1357    APPL% IFA      N/A

GOAL: RESPONSE TIME 000.00.02.000 AVG

         RESPONSE TIME EX   PERF  AVG   --- USING% --- ---------- EXECUTION DELAYS % --------- ---DLY%-- -CRYPTO%- ---CNT%--    %
SYSTEM   HHH.MM.SS.TTT VEL% INDX ADRSP   CPU  IFA  I/O  TOT  CPU                                UNKN IDLE  USG  DLY  USG  DLY QUIE

J80      000.00.01.054 57.2  0.5 151.8   0.3  N/A  0.1  0.2  0.2                                 2.8 96.7  0.0  0.0  0.0  0.0  0.0

                                  ----------RESPONSE TIME DISTRIBUTION----------
   ----TIME----    --NUMBER OF TRANSACTIONS--    -------PERCENT-------  0    10   20   30   40   50   60   70   80   90   100
   HH.MM.SS.TTT    CUM TOTAL        IN BUCKET    CUM TOTAL   IN BUCKET  |....|....|....|....|....|....|....|....|....|....|
<  00.00.01.000        36895            36895         77.7        77.7  >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
<= 00.00.01.200        37447              552         78.9         1.2  >
<= 00.00.01.400        37709              262         79.5         0.6  >
<= 00.00.01.600        38330              621         80.8         1.3  >
<= 00.00.01.800        38566              236         81.3         0.5  >
<= 00.00.02.000        38782              216         81.7         0.5  >
<= 00.00.02.200        39288              506         82.8         1.1  >
<= 00.00.02.400        39457              169         83.1         0.4  >
<= 00.00.02.600        40033              576         84.4         1.2  >
<= 00.00.02.800        40149              116         84.6         0.2  >
<= 00.00.03.000        40296              147         84.9         0.3  >
<= 00.00.04.000        42102             1806         88.7         3.8  >>>
<= 00.00.08.000        46154             4052         97.2         8.5  >>>>>
>  00.00.08.000        47460             1306          100         2.8  >>
```

Abbildung 5.15: Workload Activity Report des Workloads TSO.

Man kann in diesem Beispiel sehr exakte Informationen über den TSO-Workload erfahren:

- Activity Reports werden nach Service- oder Reportklassen gruppiert. In diesem Fall ist die Serviceklasse TSO.
- Als Ziel der Serviceklasse wurde eine durchschnittliche Antwortzeit von 2 Sekunden festgesetzt. Dieses Ziel wurde in den betrachteten 30 Minuten (Intervallzeit) mit einer Antwortzeit von durchschnittlich 1.054 Sekunden übertroffen.
- Demnach liegt der Performanceindex bei 0.5.
- Während des Intervalls wurden 36.8% der zur Verfügung stehenden CPU-Zeit verbraucht. Zu beachten ist hierbei, dass die zur Verfügung stehende CPU-Menge von der Anzahl der installierten CPUs abhängig ist. Bei einer CPU entspricht eine Vollauslastung der CPU 100%, bei zwei CPUs 200%, usw.
- Von 47460 Transaktionen haben 38782 die Zielzeit eingehalten.

- Nur 0.2% des TSO-Workloads wurden durch Delays an der CPU aufgehalten.

Die von einem Post-Processor erstellten Reports lassen sich im *RMF Spreadsheet Reporter* auf einem Windows-Rechner in einer grafischen Form darstellen.

5.4.2 RMF-PM auf z/OS

RMF-PM ist ein weiteres Frontend zur grafischen Darstellung der von Monitor III gesammelten Informationen. Im Gegensatz zum Spreadsheet Reporter, der durch den Post-Processor vorverarbeitete, historische Daten des Monitor I darstellt, stellt RMF-PM Daten dar, die im letzten Erfassungsintervall von Monitor III akkumuliert wurden. RMF-PM eignet sich also zur Online-Überwachung eines Systems.

Mit RMF-PM lassen sich folgende Performancedaten von z/OS darstellen ([KC+06]):

- Generelle Hardwarenutzung durch Jobs, wie z.B. Prozessornutzung, Speichernutzung, Belegung von I/O-Pfaden, Verzögerungen durch Zugriffskonflikte auf Ressourcen, etc.
- WLM-spezifische Daten zu Workloads, wie z.B. WLM Serviceklassen, WLM Reportklassen, Workflow, etc.

RMF-PM erhält seine Daten von einem *Distributed Data Server*, kurz *DDS*. Der DDS ruft alle Performancedaten der Sysplexmitglieder aus den SDS ab und stellt sie über eine TCP/IP-Schnittstelle dem RMF-PM-Frontend zur Verfügung.

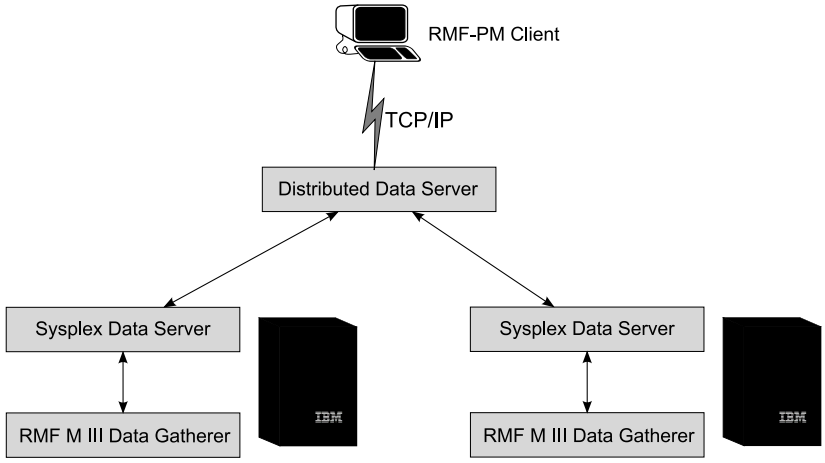


Abbildung 5.16: RMF-PM und der DDS.

5.4.3 RMF-PM für Linux

RMF-PM wurde für diese Arbeit ausgewählt, da es für beide zu untersuchende Systeme verfügbar ist. Für Linux stellt IBM ein *RMF Linux Data Gatherer*-Paket, kurz *rmfpms*, zur Installation auf dem Server zur Verfügung. rmfpms für Linux ist das Äquivalent zum DDS auf z/OS.

Unter Linux hat man betriebssystembedingt weniger Metriken als unter z/OS zur Analyse zur Verfügung. Die messbaren Metriken lassen sich in sechs Gruppen unterteilen ([CK+05]):

- System- und Prozessinformationen: Informationen über die Anzahl neu erstellter Prozesse und die Anzahl der Kontextwechsel.
- Netzwerkbelastung: Anzahl der ankommenden und ausgehenden Pakete/Bytes pro Sekunde und die Fehlerrate.
- CPU-Verbrauch: Die CPU-Belastung durch Prozesse.
- CPU-Load: Länge der Warteschlange der auf die CPU wartenden Prozesse.
- Dateisystemnutzung: Belegter und freier Festplattenplatz in Prozent und Megabyte.
- Speicherverwaltung und -belastung: Anzeige der Seitenzugriffsfehler pro Prozess. Anzeige der belegten Seiten im Speicher und Größe des virtuellen Speichers der Prozesse.

5.4.4 RMF-PM Client

Der RMF-PM Client ist für beide Serverarten derselbe. Nach dem Starten des Programms stellt man zuerst die notwendigen Verbindungen zu z/OS bzw. Linux her. Die Auswertung der Daten ist in sogenannten *PerfDesks* organisiert, die *DataViews* enthalten. PerfDesks können abgespeichert und exportiert werden, um sie wieder zu verwenden und sie können genau auf die Bedürfnisse der jeweiligen Untersuchung zugeschnitten werden.

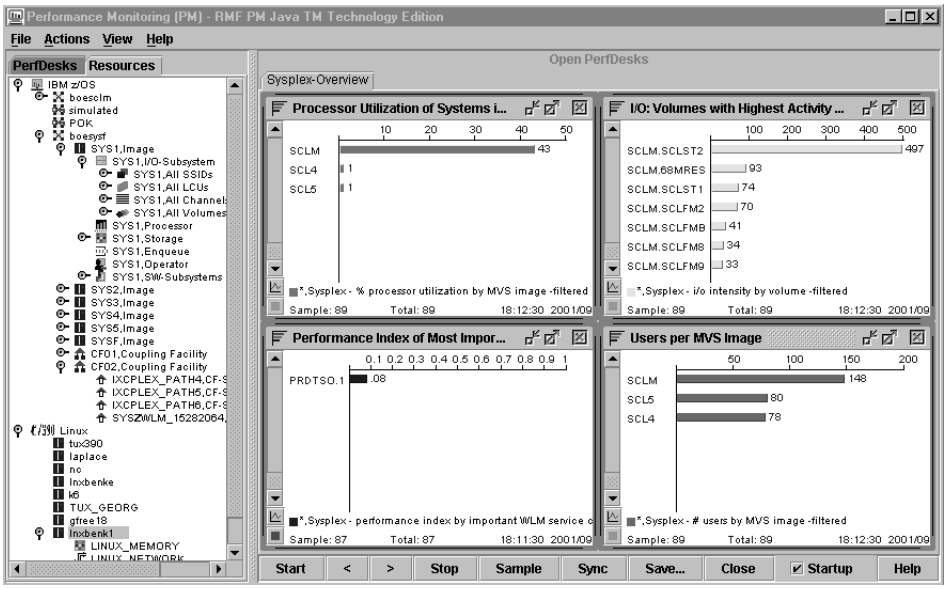


Abbildung 5.17: RMF-PM in der Anwendung.

Abbildung 5.17 zeigt eine gestartete RMF-PM-Anwendung mit einem geöffneten PerfDesk *Sysplex-Overview*. Innerhalb des PerfDesks befinden sich vier *DataViews*, die jeweils eine *Series* enthalten. Eine Series setzt sich zusammen aus einer überwachten Ressource und einer Reihe von Zeitstempeln oder einer Liste aus Name/Wert-Paaren.

Links im Bild ist der Ressourcenbaum zu sehen, der die überwachten Systeme anzeigt und aus dem Ressourcen ausgewählt, und neue PerfDesks und DataViews zur Überwachung der ausgewählten Ressourcen angelegt werden können. Für eine weitergehende Einführung in RMF-PM sei an dieser Stelle auf [CK+05] verwiesen.

Die aufgezeichneten Daten können in einem Tabellendatenformat exportiert werden, um dann in einem Tabellenkalkulationsprogramm weiter verarbeitet zu werden. So lassen sich aus den erfassten Daten anschauliche Grafiken erstellen. In dieser Arbeit werden die im nächsten Kapitel verwendeten Diagramme auf diesem Weg erstellt.

5.5 Zusammenfassung

Für einen Vergleich der integrierten und der verteilten Installation eines WebSphere Application Servers wurde im Verlauf dieser Arbeit ein Szenario entworfen, das ermöglicht die beiden WebSphere Application Server unter Belastung zu setzen und gleichzeitig den Ressourcenverbrauch zu dokumentieren.

Als Benchmarkanwendung für die beiden Anwendungsserver kommt der von IBM entwickelte Trade6-Benchmark zum Einsatz, der eine breite Verwendung der Java EE-Komponenten bietet. Apache jMeter wird benutzt, um wiederholbare Lasttests für die Trade-Anwendung zu generieren und auszuführen. Während der Tests werden Ressourcenverbräuche über RMF-PM erfasst.

Als problematisch hat sich hier die mäßige Vergleichbarkeit bzw. die großen Unterschiede in der Menge und Art der zu erfassenden Metriken für die beiden Betriebssysteme z/OS und Linux herausgestellt. z/OS bietet mit RMF eine aufwändiges und mächtiges Werkzeug zur Analyse und Erfassung von Ressourcenverbräuchen. Der Linux-Kernel lässt keine so tiefen Einblicke in das System zu.

Abbildung 5.18 stellt die beiden aufgestellten Konfigurationen zusammengefasst dar:

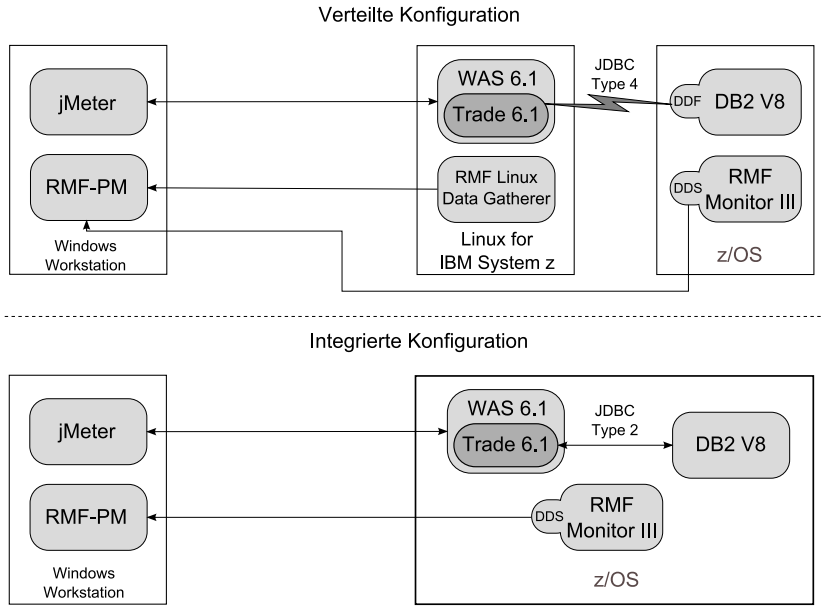


Abbildung 5.18: Die beiden Testkonfigurationen im Überblick.

6 Die Testläufe und ihre Auswertung

Im Versuchsaufbau, der im vorangegangenen Kapitel vorgestellt wurde, werden nun verschiedene Messungen durchgeführt, um den Ressourcenverbrauch in beiden Systemen unter dem Einfluss ansteigender Last zu dokumentieren. Dazu werden im Lastgenerator jMeter verschiedene Nutzungsprofile (*Use Cases*) erstellt und ausgeführt. Vorab soll aber zuerst noch allgemein geklärt werden, wie man Performancetests durchführt und welche Aspekte speziell beachtet werden müssen.

6.1 Aspekte einer Performanceuntersuchung

Es gibt verschiedene Möglichkeiten eine Anwendung zu testen ([Sch07]). Während der Entwicklung einer Anwendung werden häufig *funktionale Tests* durchgeführt, mit denen die Korrektheit der Implementierung überprüft wird. Ist die Anwendung fertig entwickelt und soll sie in das Tagesgeschäft einer Firma integriert werden, werden *Integrationstests* durchgeführt. Mit diesen wird untersucht, wie sich die Anwendung in eine bestehende Geschäftsumgebung einfügt und welche Folgen dies für andere, parallel laufende Anwendungen hat.

Zwischen diesen beiden Testphasen finden die *Performancetests* statt. Es gibt verschiedene Arten von Performancetests ([CJC06]). Einfache oder auch *primitive Tests* testen eine Anwendung auf einzelne Funktionen, wie zum Beispiel *PingServlet*, wobei serverseitig die dynamische Erstellung einer HTML-Seite getestet wird. *Client Benchmarks* führen die grundlegenden Funktionen einer Anwendung auf echten Daten aus, entsprechen aber nicht der Simulation einer der Realität entsprechenden Verwendung der Applikation. *Operationale Performancetests* sind sehr aufwändige Tests und entsprechen einem simulierten Integrationstests.

Im Folgenden werden sogenannte *End-to-End*-Tests durchgeführt ([CJC06]), in denen mit einer realitätsnahen Belastung die Anwendung vom Client bis hin zum Backend getestet wird. Trade6 ist ein Benchmark der speziell für solche Tests des WebSphere Application Server und die umgebende Infrastruktur erstellt wurde. Dabei stoßen mehrere Benutzer parallel (*concurrent users*) mehrere Geschäftstransaktionen an. Dies soll einer Belastung entsprechen, wie sie in einem realen Benutzungsszenario vorkommen könnte. Dabei ist zu beachten, dass Geschäftstransaktionen nicht technischen Transaktionen entsprechen. Geschäftstransaktionen, wie zum Beispiel die Bestellung einer Aktie in Trade6, entsprechen mehreren technischen Transaktionen ([Sch07]).

Bei der Erstellung und der Ausführung von Performancetests, im Speziellen bei End-to-End-Tests mit Trade6, müssen einige Punkte beachtet werden. Ein wichtiger Faktor bei Tests des WebSphere Application Server ist die *Java Virtual Machine*.

6.1.1 Die JVM und der JIT-Compiler

Der *Just-In-Time*-Compiler, kurz *JIT-Compiler*, der Java Runtime Environment wurde entwickelt, um das Laufzeitverhalten von Java-Anwendungen zu verbessern. Java ist eine Interpretersprache, d.h. der aus dem Quellcode vom Javacompiler (javac) erzeugte *Bytecode* in den Class-Dateien

ist eine plattformunabhängige Ausführungsbeschreibung des Programms. Zur Laufzeit der Anwendung liest die JVM den Bytecode ein, bestimmt seine Semantik (interpretiert ihn) und führt dann die entsprechende Logik aus. Dieser Interpretationsprozess verlangsamt die Ausführung einer Java-Anwendung im Vergleich zu einer nativen Anwendung, die direkt auf der CPU ausgeführt wird ([IBM07]).

Der JIT-Compiler hilft diese Ausführungszeit zu verkürzen. Wird eine Java-Methode aufgerufen, kompiliert er diese in den nativen Maschinencode der ausführenden Hardware. „Just in time" daher, da dies während der Ausführung des Codes geschieht. In der Folge wird bei einem weiteren Aufruf dieser Methode nicht mehr der Bytecode interpretiert, sondern der kompilierte Code nativ ausgeführt.

Führt man eine JVM und darauf dann eine Java-Anwendung aus, werden schon während des Starts der JVM und der darauf ablaufenden Anwendung Methoden ausgeführt. Das Kompilieren der Methoden benötigt Zeit und ein aktivierter JIT-Compiler verlängert die Anlaufzeit (*startup time*) der JVM und der darin ausgeführten Anwendung. Wäre der JIT-Compiler deaktiviert, würde die JVM schneller starten, aber die Programme innerhalb der JVM würden im weiteren Verlauf langsamer ablaufen ([IBM07]).

In der Praxis werden die Methoden nicht bei ihrem ersten Aufruf kompiliert. Für jede Methode führt die JVM einen Zähler mit, wie oft die Methode schon aufgerufen wurde. Bei jedem Aufruf wird dieser Zähler um Eins erhöht. Solange ein bestimmter Schwellenwert (*Compilation Threshold*) nicht erreicht wird, wird die Methode weiterhin interpretativ ausgeführt. Ist der Schwellenwert erreicht, wird die Methode vom JIT-Compiler kompiliert. So werden häufig aufgerufene Methoden früher kompiliert als weniger häufig aufgerufene Methoden. Der Schwellenwert wird von den Entwicklern der JVM so gewählt, dass eine optimale Balance zwischen Anlaufzeit der JVM und Laufzeitverhalten des Programms gegeben ist.

Wurde eine Methode durch den JIT-Compiler kompiliert, wird der Zähler wieder auf Null gesetzt. Weitere Methodenaufrufe erhöhen den Zähler erneut. Wird dann ein Rekompilierungs-Schwellenwert (*Recompilation Threshold*) erreicht, wird die Methode ein weiteres Mal kompiliert. Diesmal aber mit einer größeren Menge an vorgenommenen Optimierungen. Optimierungen kosten Zeit und so werden nur wirklich tragende Methoden der Anwendung, die oft aufgerufen werden, aufwändig optimiert. Dieser Prozess findet iterativ statt und kann sich mehrere Male wiederholen. So soll der Nutzen des JIT-Compiler maximiert werden ([IBM07]).

Der JIT-Compiler optimiert den Code anhand einer internen Repräsentation des Bytecodes als Baumstruktur (tree), die eher dem Maschinencode entspricht, als der Bytecode. Anhand dieser Baumstruktur finden die Analyse und die Optimierung des Codes statt. Folgende, aus dem Compilerbau bekannte, Optimierungen und Schritte werden dabei auf der Baumrepräsentation durchgeführt:

- **Inlining:** Hierbei werden Methodenaufrufe durch die Methode selbst ersetzt.
- **Lokale Optimierungen:** Hier werden kleine Abschnitte mittels Datenflussanalyse und Optimierung der Registernutzung verbessert.
- **Kontrollflussoptimierung:** Hierbei wird der Kontrollfluss (*control flow*) analysiert und Codepfade (*code paths*) neu angeordnet. Optimierungen dieser Art umfassen zum Beispiel *loop unrolling* oder das Entfernen überflüssigen Codes.

- **Globale Optimierungen:** Globale Optimierungen betreffen die gesamte Methode und beinhalten zum Beispiel das Entfernen von redundantem Code oder Optimierungen der Speicherallokation. Nach der globalen Optimierung werden häufig wieder lokale Optimierungen durchgeführt.
- **Erstellung des nativen Codes:** Aus der Baumrepräsentation und mit den darauf angewandten Optimierungen wird der native Maschinencode kompiliert.

Bis auf den letzten Schritt sind die Teilschritte unabhängig von der darunter liegenden Hardware und für alle Plattformen gleich.

Möchte man Performance testen, ist der JIT-Compiler ein wichtiger Aspekt, den es zu beachten gilt. Man muss der zu testenden Anwendung eine gewisse Vorlaufzeit geben, damit der JIT-Compiler wirken kann und man von den Performanceverbesserungen durch ihn profitiert. Die Trade6-Entwickler empfehlen jeden Codepfad in Trade6 mindestens 3000 Mal auszuführen ([IBM06]) bevor mit Testläufen begonnen wird. Daher werden vor den Testläufen die Tests mehrfach ohne zu Messen durchgeführt, um so dem JIT-Compiler die Möglichkeit zu geben, die Anwendung im Laufzeitverhalten zu optimieren.

6.1.2 Trade6 - Beschreibung und Konfiguration

Ein End-to-End-Test soll die Belastung durch eine reale Benutzung simulieren. Zu diesem Zweck muss im Lastgenerator eine Reihe von Aktionen definiert werden, die einer realen Nutzung der Anwendung entsprechen.

Zu beachten ist hierbei, dass sich im Realfall nicht alle Anwender gleich verhalten, d.h. zu allererst gilt es die Anwender zu kategorisieren. Es wird angenommen, dass sich die Nutzer der Trade6-Anwendung in drei Klassen unterscheiden lassen ([KC$^+$06]):

- **Browsing user:** Benutzer, die den aktuellen Stand ihrer Aktien und ihres Accounts überprüfen.
- **Buying user:** Benutzer, die Aktien kaufen.
- **Selling user:** Benutzer, die Aktien verkaufen.

Es wird weiterhin angenommen, dass die Zahl der Benutzer, die lediglich in ihrem Account stöbern, größer ist, als die Zahl der Benutzer, die Aktien kaufen oder verkaufen ([KC$^+$06]). Die Gesamtzahl der Nutzer wird in den Tests nach dem Verhältnis 3:1:1 auf die Klassen verteilt, d.h. es greifen immer drei mal mehr Benutzer (fast nur) lesend auf die Trade6-Anwendung zu, als schreibend.

Die Nutzer der drei Klassen führen während der Lasttests folgende Aktionen aus:

- **Browsing user**:
  - Aufruf der Loginseite
  - Login
  - Aufruf des Portfolios
  - Prüfen des Accounts

  - Wechsel zur Home-Seite
  - Logout

- **Buying user**:
  - Aufruf der Loginseite
  - Login
  - Anzeigen elf zufällig ausgewählter, zum Kauf bereitstehender Aktien
  - Kauf einer zufällig ausgewählten Menge einer der zuvor angezeigten Aktien
  - Anzeigen des eigenen Portfolios zur Überprüfung des Kaufs
  - Logout

- **Selling user**:
  - Aufruf der Loginseite
  - Login
  - Wechsel zum Portfolio
  - Verkauf einer der im Besitz des Benutzer befindlichen Aktie
  - Anzeigen des Portfolios, um den Verkauf zu überprüfen
  - Logout

Die Anzahl der gleichzeitig auf die Anwendung zugreifenden Benutzer wird pro Testlauf sukzessive nach folgendem Schema erhöht:

Benutzer insgesamt	Browsing user	Buying user	Selling user	Loops
50	30	10	10	67
100	60	20	20	34
150	90	30	30	23
200	120	40	40	17
250	150	50	50	14
300	180	60	60	12

In jMeter wird für jeden Benutzer ein Thread erstellt. Die Anzahl der Loops gibt dabei an, wie oft jeder Thread die für ihn definierten Aktionen wiederholt. Das bedeutet, ein Thread arbeitet die Liste der ihm zugewiesenen Aktionen der Reihe nach ab und beginnt dann wieder von vorne. Wie oft dieser Neuanfang stattfindet bestimmt die Anzahl der Loops. Die Loops wurden so berechnet, dass im Endeffekt pro Testlauf rund 20000 Requests an den Server gestellt werden.

Dieses Anwendungsverhalten gilt es nun in jMeter zu simulieren.

6.2 Die Konfiguration des Lastgenerators jMeter

Die drei Nutzerklassen werden in jMeter durch drei Threadgruppen simuliert, je eine für Browsing, Buying und Selling User. In den drei Threadgruppen werden nun verschiedene *Sampler*, *Logic Controllers*, *Configuration Elements*, *Listener* und *Timer* konfiguriert, um die oben beschriebenen Anwender nachzuahmen.

Timer

Timer simulieren die Zeit, die ein Benutzer eine Seite betrachtet, bevor er den nächsten Link aktiviert (engl. *think time*). Ein Benutzer schaut sich die erhaltene Antwortseite zuerst an und analysiert sie, bevor er eine neue Seite anfordert. Möchte man reales Benutzerverhalten simulieren, so muss man dies beachten, und in die Erstellung der Tests mit einbeziehen.

Der in den Testläufen verwendete *Gaussian Random Timer* besitzen eine feststehende Dauer zu der ein zufällig ausgewählter Zeitraum hinzu addiert wird. In den nachfolgenden Tests werden als fixe Dauer 2000ms und als variabler Zeitraum 3000ms verwendet. Vor jedem Sampler wird dadurch zwischen 2s und 5s gewartet, bevor er dann ausgeführt wird.

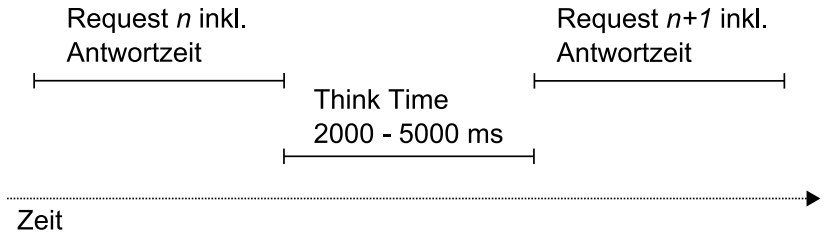


Abbildung 6.1: Auswirkung der Think Time.

Die Think Time ist kurz gehalten, um die Gesamtdauer eines Testlaufs ebenfalls kurz zu halten.

Configuration Elements

Configuration Elements ermöglichen das Einrichten von Standardeinstellungen und Variablen, die für die ganze Threadgruppe oder einen anderen Gültigkeitsbereich von Bedeutung sind.

Um nicht bei jedem Sampler die vollen Serverinformationen eingeben zu müssen, gibt es das *HTTP Request Defaults*-Element. Hier können Standardeinstellungen, wie IP-Adresse des Servers, Portnummer der Anwendung und Pfad zur Anwendung, angegeben werden, die dann für die gesamte Threadgruppe gelten. So kann man sich in den eigentlichen Samplern auf die Angabe der durchzuführenden Aktion beschränken.

Trade6 führt die Sitzungsverwaltung auf Basis von Session-IDs durch. Diese werden mit dem Einlogvorgang für jede Sitzung erstellt und dem Benutzer in einem Cookie mitgeteilt.Der verwendete Lastgenerator muss Cookie-Verwaltung beherrschen. Das *HTTP Cookie Manager*-Element von jMeter verwaltet Cookies wie ein Webbrowser. Jeder Thread hat dabei seinen eigenen Cookie-Speicher in Form einer Threadvariablen, so dass die einzelnen Sessions von anderen Threads abgekapselt sind ([jMe08]).

Listener

Listener zeichnen Daten während der Testläufe auf und stellen sie dem Tester zur Verfügung. In jeder Threadgruppe wird ein *View Result Tree*-Listener konfiguriert, der die Requests und dazugehörigen Server-Responses im XML-Format speichert und darstellt. jMeter beherrscht das Rendern von HTML- und XML-Seiten. So kann jeder Request mit seiner zugehörigen Antwortseite im Nachhinein nachvollzogen werden.

Unabhängig von einer Threadgruppe wird noch ein globaler *Aggregate Report*-Listener konfiguriert, der statistische Daten über alle Requests gemittelt darstellt. Auf ihn wird zu einem späteren Zeitpunkt noch genauer eingegangen (S.99 ff.).

Die bis hierhin vorgestellten jMeter-Elemente werden in allen Threadgruppen verwendet. Die Threadgruppen für die einzelnen Nutzerklassen unterscheiden sich nur in der Form der Sampler.

Sampler

Die Sampler lösen die eigentlichen Aktionen in der zu testenden Anwendung aus. Sie simulieren die eigentlichen Aktionen der Benutzer. Manche Sampler sind in allen Threadgruppen vorhanden, da manche Aktionen von allen Benutzern durchgeführt werden müssen:

- **Get Loginpage:** Bevor sich ein Benutzer einloggen kann, muss er zuerst die Login-Seite des Trade6-Benchmarks anfordern. Der Sampler führt keine spezifische Aktion aus, sondern ruft lediglich die URL auf, die in den HTTP Request Defaults definiert ist.

```
http://*:*/trade/app
```

- **Login:** Jeder Benutzer muss sich vor seiner Verwendung zuerst beim Trade6-Benchmark anmelden. Für ein realistisches Testszenario ist es wichtig, dass sich die Anwender während des Tests auf die gesamte Nutzerbasis verteilen. Es wäre falsch nur einige wenige Benutzer oder sogar nur einen in einem Performancetest zu verwenden ([Sch07]). Um dies auszuschließen wird jMeter wie folgt konfiguriert:

  Zuerst wird ein Sampler konfiguriert, der dem http-Request

```
http://*:*/trade/app?action=login&uid=<UserID>&passwd=xxx
```

  entspricht.

  Das Passwort ist für alle Nutzer gleich (*xxx*). Um unterschiedliche Nutzer zu konfigurieren, wird ein *CSV Data Set Config*-Konfigurationselement verwendet. Mit ihm lässt sich eine CSV-Datei auslesen, in dem die User-IDs in einer einfachen Tab-getrennten Liste stehen. Für jeden neuen Thread wird aus der Liste eine neue User-ID ausgelesen. Sollte das Ende der Datei erreicht werden, so wird über die Option *Recycle on EOF?* wieder am Anfang der Datei begonnen User-IDs auszulesen. Somit wird zu keinem Zeitpunkt des Tests eine User-ID von zwei Threads gleichzeitig verwendet, solange die Anzahl der gleichzeitigen Benutzer nicht die Anzahl der zur Verfügung stehenden User-IDs übersteigt.

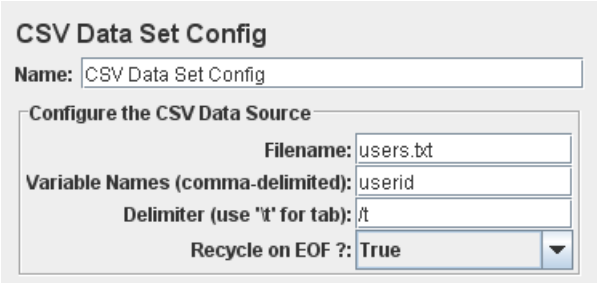


Abbildung 6.2: jMeter-Screenshot - CSV Data Set Config

Mit der so erhaltenen User-ID wird ein Sampler konfiguriert, der den Login des Benutzers ausführt.

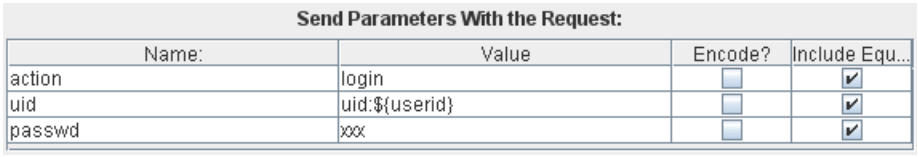


Abbildung 6.3: jMeter-Screenshot - Login-Action und zugehörige Parameter.

- **Logout:** Ein einfacher Sampler mit der Aktion *logout*.

```
http://*:*/trade/app?action=logout
```

Zwischen dem Ein- und Ausloggen eines Benutzers unterscheiden sich nun die durchgeführten Aktionen je nach Nutzerklasse.

Sampler: Browsing User

Diese Anwender führen nach dem Login drei Aktionen durch, welche den folgenden drei parameterlosen HTTP-Requests entsprechen. Für jede Anfrage ist jeweils ein Sampler in der Threadgruppe *Browsing User* konfiguriert (vgl. S.66):

```
http://*:*/trade/app?action=portfolio
http://*:*/trade/app?action=account
http://*:*/trade/app?action=home
```

Abbildung 6.4 zeigt die gesamte Threadgruppe Browsing User in jMeter.

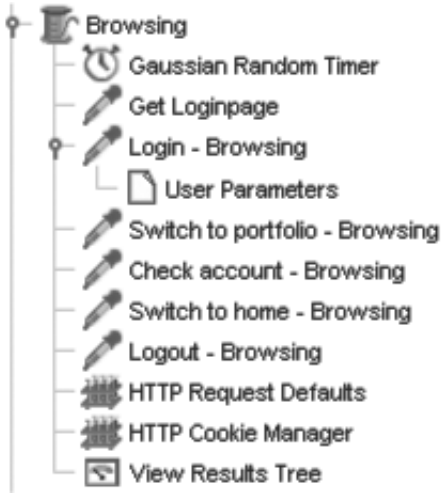


Abbildung 6.4: Die Threadgruppe für die Klasse Browsing User.

Sampler: Buying User

Der kaufende Benutzer lässt sich zuerst eine Auswahl zu kaufender Aktien anzeigen. In diesem Sampler wird die `__Random`-Funktion von jMeter verwendet um die anzuzeigenden Aktien zufällig aus der gesamten Aktienbasis auszuwählen. Funktionen geben einem in jMeter die Möglichkeit auszuführende Logik in den Samplern zu verwenden und sie werden direkt in den Request eingefügt. Die Random-Funktion hat drei Parameter. Die ersten beiden geben die untere und obere Grenze des Intervalls an, aus dem eine Zahl zufällig ausgewählt wird. Der dritte Parameter ist ein Bezeichner, der verwendet werden kann, um die erhaltene Zufallszahl in der Folge anzusprechen.

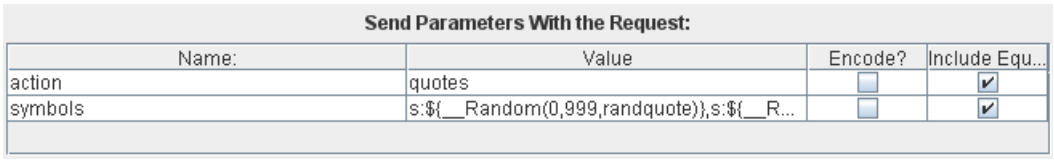


Abbildung 6.5: jMeter-Screenshot - Request um 11 zufällig ausgewählte Aktien anzuzeigen.

Der daraus resultierende HTTP-Request hat folgende Form:

```
http://*:*/trade/app?action=quotes&symbols=<StockIDs>
```

Im nächsten Sampler wird eine Aktie in zufälliger Anzahl von diesem Benutzer gekauft. In der Random-Funktion hat man die Möglichkeit sich die errechnete Zufallszahl in einer Variablen zu speichern. Diese Variable wird nun in diesem Sampler verwendet, um die im Sampler zuvor zuletzt angezeigte Aktie zu kaufen. Die Random-Funktion findet wieder Verwendung, um zufallsgesteuert die Anzahl der zu kaufenden Aktien zu errechnen.

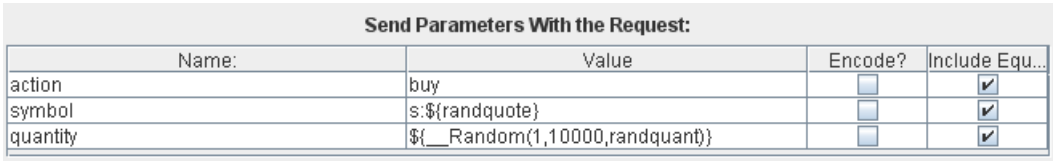


Abbildung 6.6: jMeter-Screenshot - Kauf einer Aktie.

Daraus ergibt sich der HTTP-Request:

```
http://*:*/trade/app?action=buy&symbol=<StockID>&quantity=<Anzahl>
```

Ein aktienkaufender Benutzer kontrolliert danach noch einmal sein Portfolio, um den Kauf zu überprüfen und loggt sich dann aus. Abbildung 6.7 zeigt die gesamte Threadgruppe, mit der kaufende Benutzer simuliert werden.

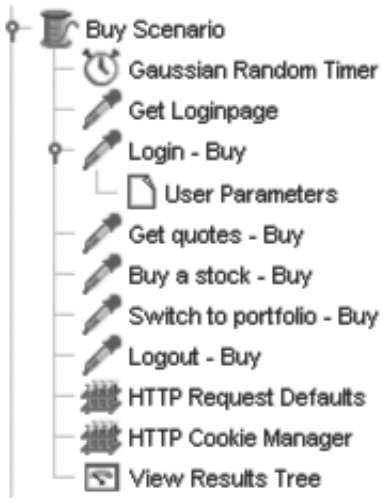


Abbildung 6.7: Die Threadgruppe für die Klasse Buying User.

Sampler: Selling User

Der verkaufende Anwender wechselt nach seinem Login zuerst zu seinem Portfolio, um sich die in seinem Besitz befindlichen Aktien anzeigen zu lassen:

```
http://*:*/trade/app?action=portfolio
```

Danach muss eine der Aktien herausgefiltert werden, die sich in seinem Besitz befindet, da die Stock-ID einer Aktie für den darauffolgenden Sell-Request benötigt wird. Hierzu wird der Post-Prozessor *Regular Expression Extractor* von jMeter verwendet. Post-Prozessoren verarbeiten Responsedaten in ihrem Gültigkeitsbereich. Möchte man den Post-Prozessor nur auf eine Antwortseite, und nicht alle zuvor in der Threadgruppe erhaltenen Seiten, anwenden, so muss der Gültigkeitsbereich des Post-Prozessors eingeschränkt werden. Dies kann mit einem *Simple Controller* realisiert werden. Dieser besitzt keine Funktionalität, sondern dient lediglich zur Beschränkung des Gültigkeitsbereichs.

In diesem beschränkten Gültigkeitsbereich findet dann der Aufruf des Portfolios (s.o.) statt und anschließend die Auswertung der Antwortseite mit dem Post-Prozessor.

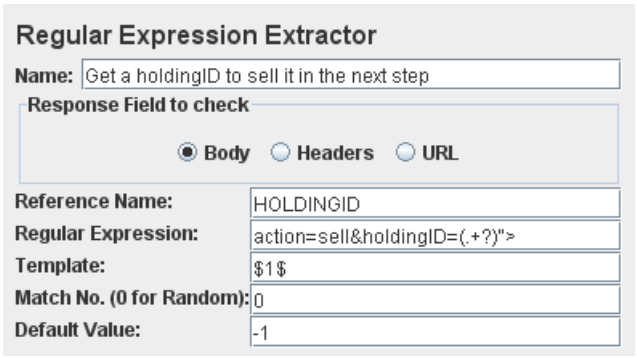


Abbildung 6.8: jMeter-Screenshot - Extrahieren eines Patterns aus einer Antwortseite.

In der Portfolio-HTML-Seite, die auf den Portfolio-Request hin vom Server zurückgeliefert wird, findet sich zu jeder im Besitz des Benutzers befindlichen Aktie ein Tabellenfeld folgender Form:

```
<TD><B><A href="app?action=sell&holdingID=408396">sell</A></B></TD>
```

Im Link findet sich die Holding-ID einer Aktie, die der Benutzer in seinem Portfolio hat. Daher wird der Post-Prozessor wie folgt konfiguriert:

- **Response Field to Check:** Der Suchbereich wird auf einen Teil der Antwortseite eingeschränkt, hier der Body der HTML-Seite.
- **Reference Name:** Dem extrahierten String wird ein Name zugewiesen. Unter diesem Namen kann im weiteren Verlauf auf das Ergebnis der Extraktion zugegriffen werden.
- **Regular Expression**: Der zu suchende reguläre Ausdruck. Der zu extrahierende Teil wird in Klammern ( ) eingeschlossen. Die darin eingeschlossene Zeichenkette „.+?“ hat Steuerfunktion. Der . gibt an, dass nach beliebigen Zeichen gesucht wird. Das +-Zeichen gibt an, dass nach einem oder mehr Zeichen gesucht wird. Das ? beendet den Suchvorgang beim ersten Treffer.

  Die Syntax der Mustersuche in jMeter entspricht der Syntax von regulären Ausdrücken der Skriptsprache Perl.
- **Template:** Das Template wird nur benötigt, wenn man mehrere Strings extrahieren möchte. Die Angabe `$1$` gibt an, dass man den gefundenen String der Gruppe Eins zuweist. Das Ergebnis der Suche ist dann unter `HOLDINGID_g1` referenzierbar.
  Würde man nach zwei Strings suchen und als Template `$1$$2$` angeben, so hätte man in der Variable `HOLDINGID_g0` beide Strings konkateniert, in `HOLDINGID_g1` den ersten und in `HOLDINGID_g2` den zweiten gefunden String.
- **Match No.:** Falls mehrere gültige Muster gefunden wurden, kann man hier auswählen, welches Muster man verwenden möchte. Da hier nach dem ersten gültigen Muster die Suche gestoppt wird, kann hier 0 für eine zufällige Auswahl aus allen Treffern angegeben werden. Es gibt in diesem Fall nur eines.
- **Default Value:** Der Wert, welcher der `HOLDINGID`-Variablen zugewiesen wird, wenn kein gültiges Muster gefunden wurde. Er ist von Bedeutung für den nachfolgenden Sell-Request.

Es kann vorkommen, dass der Benutzer keine Aktien besitzt, da er nie welche hatte oder im Verlauf des Tests alle Aktien verkauft wurden. Daher hängt die Ausführung des Sell-Samplers davon ab, ob der Wert der Variablen `HOLDINGID_g1` größer -1 ist. Nur dann wurde bei der Suche nach einem Kandidaten für den Verkauf auch einer gefunden.

Daher wird vor dem Sell-Sampler noch ein *If-Controller* eingefügt, der die `HOLDINGID`-Variable prüft.

Abbildung 6.9: jMeter-Screenshot - If-Controller.

Nur wenn die Auswertung dieser Bedingung den Wahrheitswert *True* ergibt, wird der Sell-Sampler unter Verwendung des zuvor extrahierten Strings auch ausgeführt.

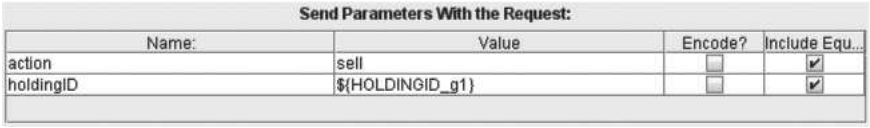


Abbildung 6.10: jMeter-Screenshot - Sampler für den Verkauf einer Aktie.

Im Abschluss an den Verkauf wechselt dieser Benutzer noch einmal zu seinem Portfolio, um den erfolgreichen Verkauf der Aktie zu kontrollieren. Abbildung 6.11 zeigt die gesamte Threadgruppe, mit der verkaufende Benutzer simuliert werden.

Abbildung 6.11: Die Threadgruppe für die Klasse Selling User.

6.3 Betrachtung der Prozessorauslastung

Im Fokus dieser Untersuchung steht die Prozessorauslastung, die durch den WebSphere Application Server und die verwendete Datenbank, in den beiden Konfigurationen entsteht. Während der

durchgeführten Lasttests wird in beiden Konfigurationen die Prozessorauslastung mittels RMF-PM aufgezeichnet. Die dabei erfassten Metriken sind:

- **% total utilization (z/OS)**: Diese Metrik gibt die durchschnittliche Auslastung des Prozessors im eingestellten Intervall (60s) aus Sicht des Betriebssytems an.
- **% cpu total active time (zLinux)**: Die durchschnittliche Auslastung des Prozessors im eingestellten Intervall (60s), ebenfalls aus Sicht des Betriebssystems.

Die Sichtweise des Betriebssystems wird betont, da beide Betriebssysteme virtualisiert ablaufen, d.h. die dediziert zugewiesenen Prozessoren werden auch noch durch den jeweiligen Hypervisor der LPARs bzw. der z/VM belastet. Die zusätzliche Last durch das Virtualisieren bewegt sich im einstelligen Prozentbereich ($<= 5\%$, [vB07]), da sonst keine anderen Betriebssysteme die Prozessoren verwenden. Der Overhead durch die Virtualisierung entsteht durch die Kontextwechsel zwischen virtueller Maschine und Gastbetriebssystem.

Beide oben genannte Metriken geben in Prozent an, zu welchem Anteil die CPU im letzten Messintervall nicht im Zustand *Idle* war, die CPU also durch ablaufende Programme benutzt wurde. Im Folgenden werden die mit RMF-PM erfassten Daten als Excel-Diagramme präsentiert. Für die Betrachtung der Prozessorauslastung werden immer zwei Diagramme ausgewertet, je eines für eine Konfiguration. Im Falle der integrierten Konfiguration wird nur die Auslastung des Prozessors der z/OS-LPAR erfasst. Für die verteilte Konfiguration wird die Prozessorauslastung beider LPARs erfasst und in einem Diagramm zusammengefasst dargestellt. Die x-Achse zeigt dabei den zeitlichen Verlauf des Tests, während auf der y-Achse prozentual die verbrauchte CPU-Zeit abgetragen ist.

Im Folgenden werden zuerst die Ergebnisse mit kurzen Kommentaren vorgestellt und in einer anschließenden Diskussion genauer erläutert. In der ersten Testreihe ist für das integrierte Szenario nur ein Prozessor konfiguriert. Hierdurch hat das integrierte Szenario einen leistungstechnischen Nachteil, welcher in einer zweiten Testreihe ausgeglichen wird (S. 91 ff.).

Testlauf mit 50 gleichzeitigen Benutzern

Diagramm 6.12 zeigt einen Testlauf mit 50 gleichzeitigen Benutzern im integrierten Szenario und einem dedizierten Prozessor für die z/OS-LPAR. Auf der x-Achse ist die Zeit in Minuten abgetragen. Die y-Achse quantifiziert die Auslastung der CPU über die Zeit in Prozent.

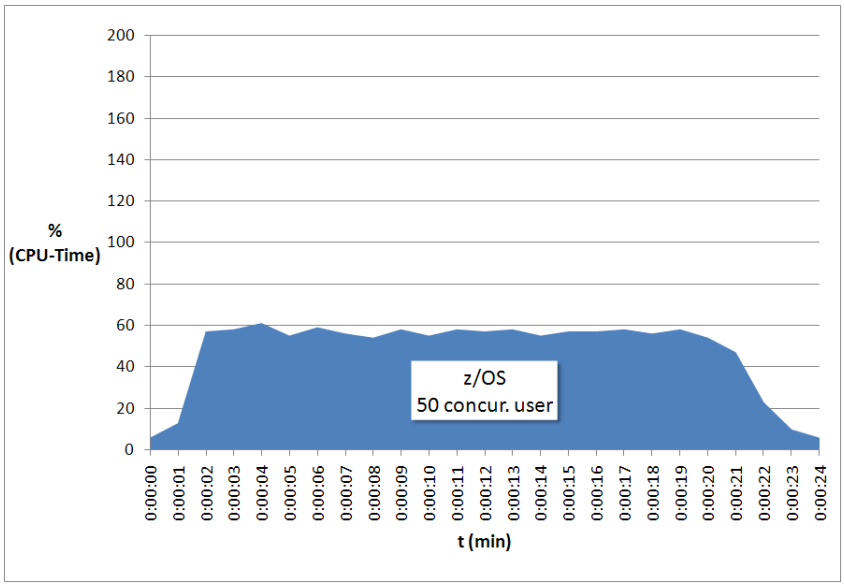


Abbildung 6.12: Prozessorauslastung - 50 gleichzeitige Benutzer - integriertes Szenario mit einem dedizierten Prozessor.

Der Prozessor der z/OS-LPAR ist durch diese Belastung zu knapp 55% ausgelastet. Diagramm 6.13 zeigt einen Testlauf mit 50 gleichzeitigen Benutzern im verteilten Szenario. Es wird die Auslastung beider Prozessoren der z/OS- bzw. zLinux-LPAR aufsummiert dargestellt.

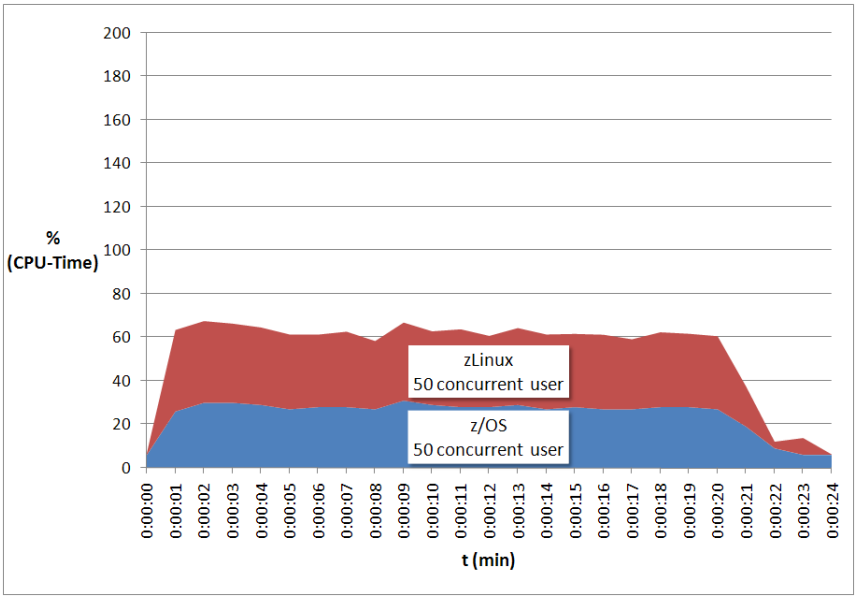


Abbildung 6.13: Prozessorauslastung - 50 gleichzeitige Benutzer - verteiltes Szenario

Die Summe der Auslastung der beiden Prozessoren ist im Vergleich zur integrierten Konfigura-

tion ca. 6 - 10% höher.

Testlauf mit 100 gleichzeitigen Benutzern

In diesem Testlauf steigt die Belastung auf 100 gleichzeitige Benutzer. Diagramm 6.14 zeigt die Prozessorauslastung im integrierten Szenario mit einem dedizierten Prozessor für die z/OS-LPAR.

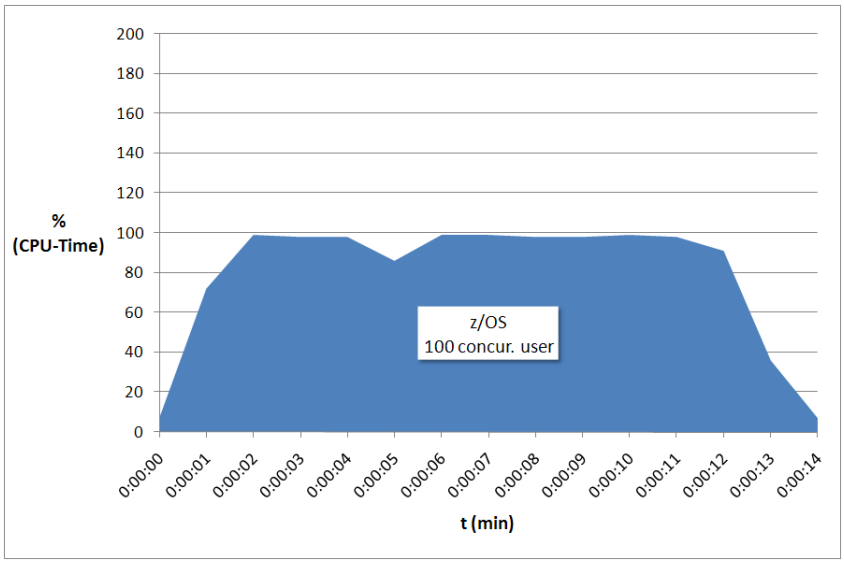


Abbildung 6.14: Prozessorauslastung - 100 gleichzeitige Benutzer - integriertes Szenario mit einem dedizierten Prozessor.

Das integrierte Szenario mit einem Prozessor ist hier schon fast saturiert, wenngleich die Skalierung sehr schön verläuft und eine doppelte Last hier die Prozessorbelastung, im Vergleich zum Lasttest mit 50 gleichzeitigen Benutzern, nicht verdoppelt. Diagramm 6.15 zeigt die Prozessorauslastung im selben Testlauf, diesmal im verteilten Szenario.

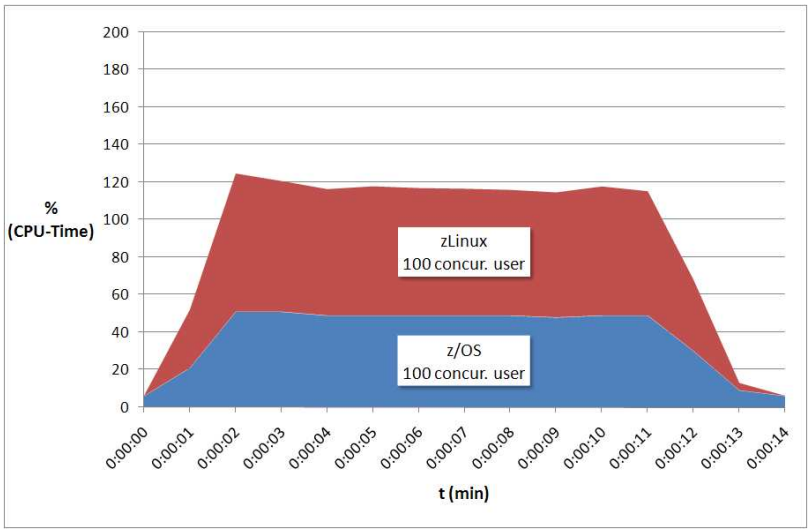


Abbildung 6.15: Prozessorauslastung - 100 gleichzeitige Benutzer - verteiltes Szenario.

Das verteilte Szenario skaliert mit 100 Benutzern ebenfalls gut und die CPU-Auslastung verdoppelt sich bei doppelter Belastung. Der Overhead durch die verteilte Architektur erhöht sich auf 10 - 20%.

Testlauf mit 150 gleichzeitigen Benutzern

Die Zahl der gleichzeitig auf die Benchmarkanwendung zugreifenden Benutzer wird für den folgenden Testlauf auf 150 erhöht. Diagramm 6.16 zeigt wieder zuerst die CPU-Auslastung über die Zeit im integrierten Szenario. Diagramm stellt 6.17 darauffolgend das Äquivalent für das verteilte Szenario dar.

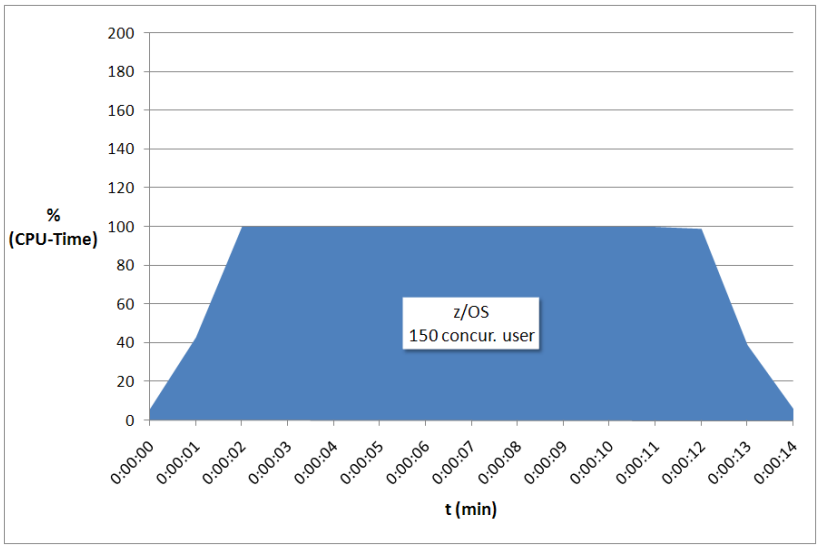


Abbildung 6.16: Prozessorauslastung - 150 gleichzeitige Benutzer - integriertes Szenario mit einem dedizierten Prozessor.

Das integrierte Szenario mit einem Prozessor hat mit 150 gleichzeitigen Benutzern seine Kapazitätsgrenze erreicht. Das zeigt sich in einer Erhöhung der Antwortzeiten pro Request, auf die später genauer eingegangen wird (S. 101 ff.). Es werden aber auch bei Überlastung immer noch alle Requests vom Server beantwortet.

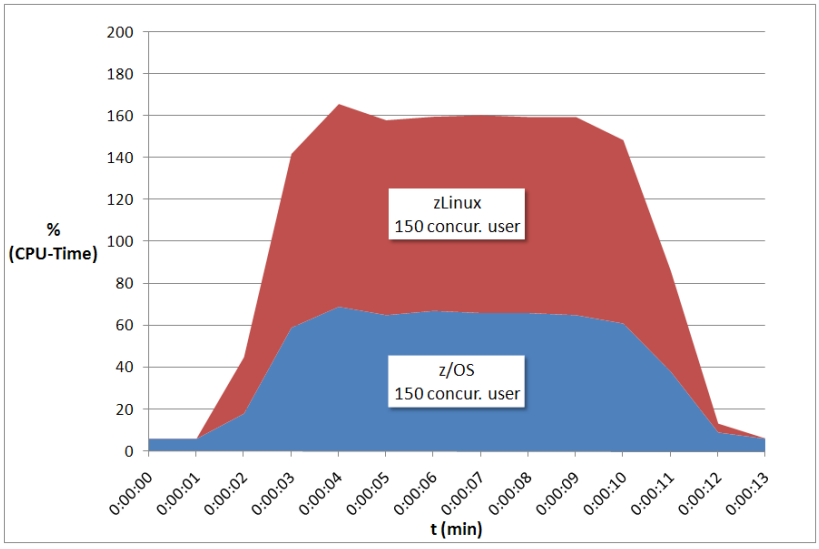


Abbildung 6.17: Prozessorauslastung - 150 gleichzeitige Benutzer - verteiltes Szenario.

Das verteilte Szenario hat durch die Verwendung von zwei Prozessoren und die Verteilung der Last noch weitere Kapazitäten zur Verfügung und ist mit 150 gleichzeitigen Benutzern noch nicht ausgelastet. Allerdings steht das zLinux-System mit dem Anwendungsserver kurz vor der

Saturierung. Die Belastung auf der zLinux-LPAR bei 150 gleichzeitigen Benutzern liegt bei ca. 95%. Rechnet man hierzu noch den Overhead durch die LPAR-Verwaltung des Hypervisor, ist der Prozessor hier schon komplett ausgelastet.

Testlauf mit 200 gleichzeitigen Benutzern

Die Diagramme 6.18 und 6.19 zeigen den Verlauf der CPU-Zeit für 200 gleichzeitige Benutzer im integrierten Szenario mit einem dedizierten Prozessor bzw. im verteilten Szenario.

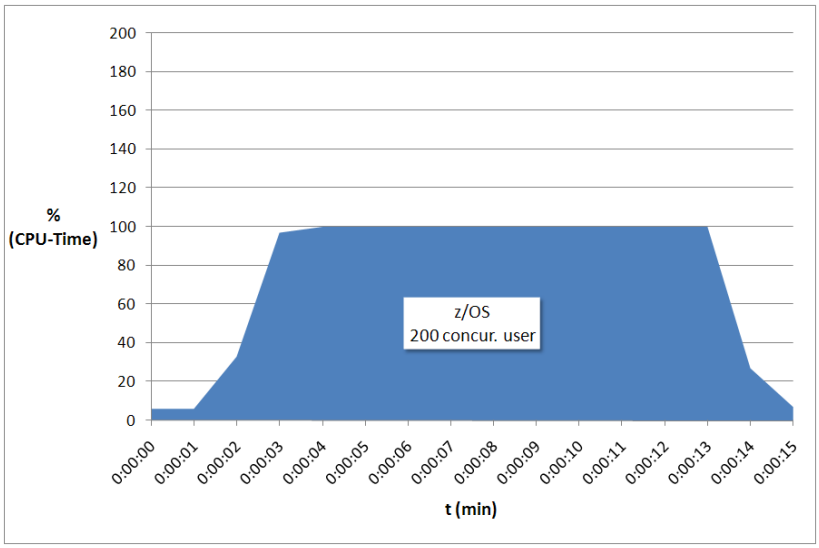


Abbildung 6.18: Prozessorauslastung - 200 gleichzeitige Benutzer - integriertes Szenario mit einem dedizierten Prozessor.

An der Gesamtauslastung des Systems ändert sich hier nichts mehr. Der Prozessor kann nicht über 100% hinaus ausgelastet werden. Die Laufzeit des Tests hat sich hier allerdings im Vergleich zu 150 gleichzeitigen Benutzern von 12min 52s auf 13min 18s erhöht. Das bedeutet, die Überlastung des Systems zeigt sich in einer Erhöhung der einzelnen Antwortzeiten der Requests, welche sich in der Erhöhung der Gesamtdauer des Testlaufs widerspiegelt.

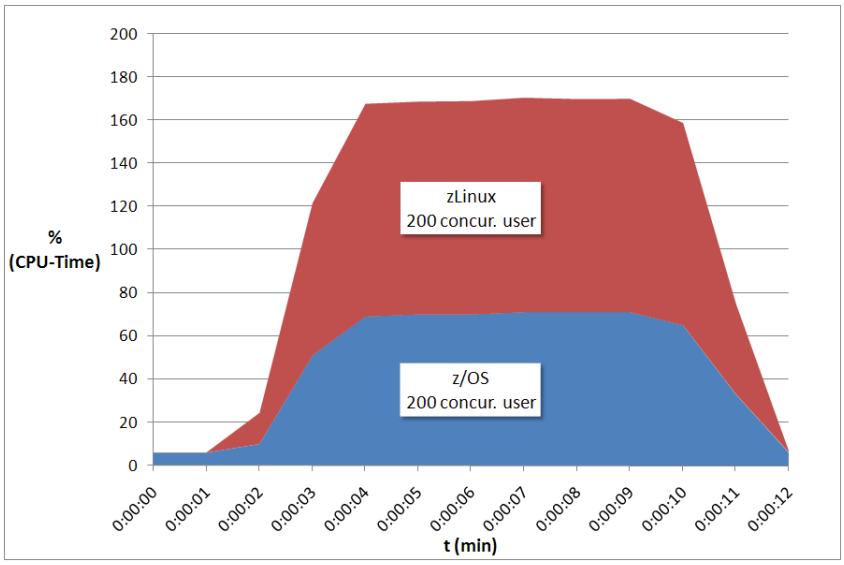


Abbildung 6.19: Prozessorauslastung - 200 gleichzeitige Benutzer - verteiltes Szenario.

Mit 200 gleichzeitigen Benutzern ist jetzt auch die zLinux-Partition mit dem Anwendungsserver voll ausgelastet. Die Auslastung der z/OS-Partition im verteilten Szenario liegt durch die entfernten Datenbankzugriffe bei ca. 70%. Das bedeutet, das verteilte Szenario ist nicht in der Lage die gegebenen Prozessorressourcen voll auszunutzen. Der Anwendungsserver kann nicht mehr Anfragen an die Datenbank stellen, als der Prozessor der zLinux-LPAR zulässt.

Testlauf mit 250 gleichzeitigen Benutzern

In den Testläufen, für die in den Diagrammen 6.20 und 6.21 der zeitliche Verlauf der CPU-Zeit dargestellt ist, wird die Anzahl gleichzeitiger Benutzer auf 250 erhöht.

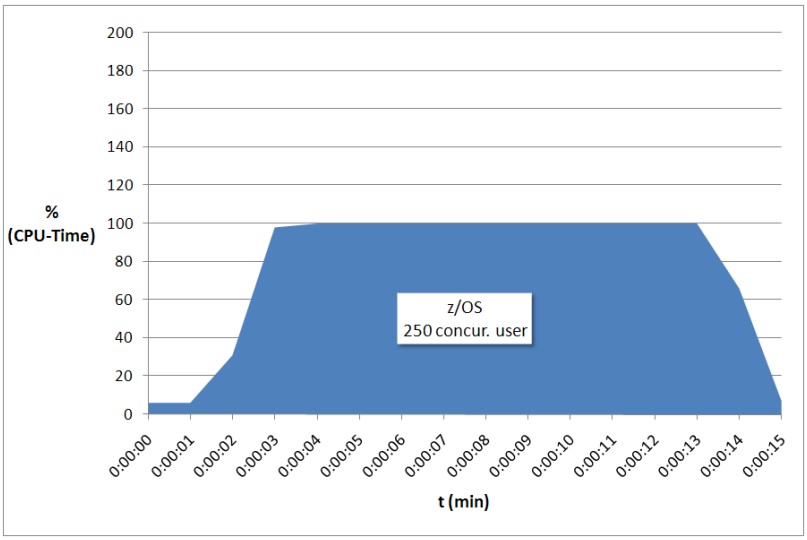


Abbildung 6.20: Prozessorauslastung - 250 gleichzeitige Benutzer - integriertes Szenario mit einem dedizierten Prozessor.

Die Testlaufzeit hat sich hier von 13min 18s bei 200 gleichzeitigen Benutzern auf 13min 55s erhöht. Das System ist deutlich überlastet.

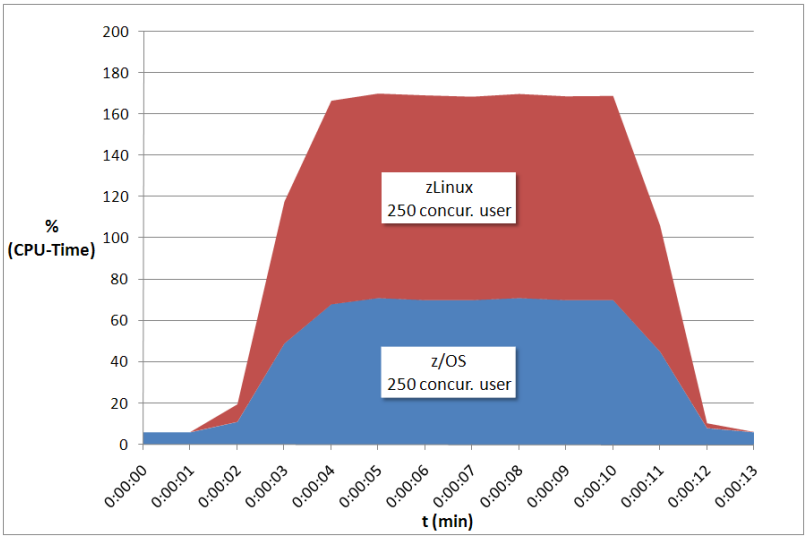


Abbildung 6.21: Prozessorauslastung - 250 gleichzeitige Benutzer - verteiltes Szenario.

Abbildung 6.21 ist fast identisch zu Abbildung 6.19. Die zLinux-Partition ist saturiert und kann die Leistung nicht mehr erhöhen. Die z/OS-Partition liegt die Prozessorleistung betreffend zu 30% brach.

Testlauf mit 300 gleichzeitigen Benutzern

Mit 300 gleichzeitigen Benutzern endet die Testreihe. Beide Konfigurationen sind jetzt überlastet, wie die Diagramme 6.22 bzw. 6.23 zeigen. In Unterkapitel 6.4.2 sieht man die Auswirkung der Überlastung an den schlechter werdenden Antwortzeiten der Requests.

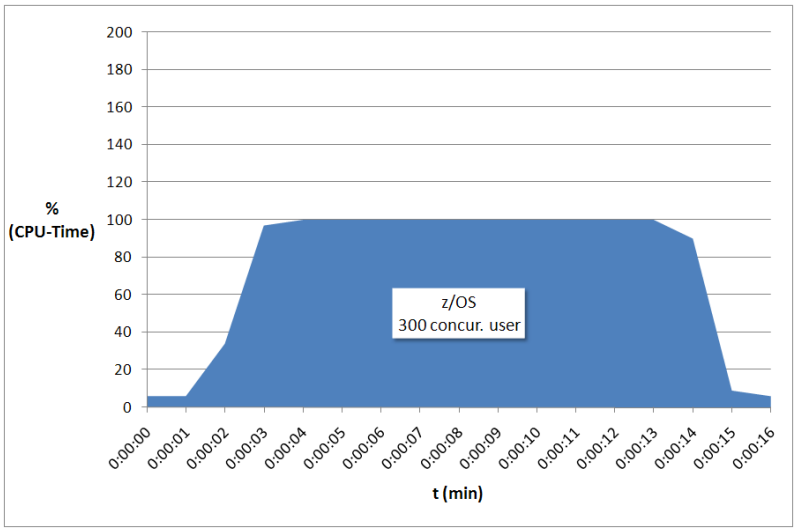


Abbildung 6.22: Prozessorauslastung - 300 gleichzeitige Benutzer - integriertes Szenario mit einem dedizierten Prozessor.

Die Testlaufzeit erhöht sich nochmals auf 14min 3s. Das System ist stark überlastet.

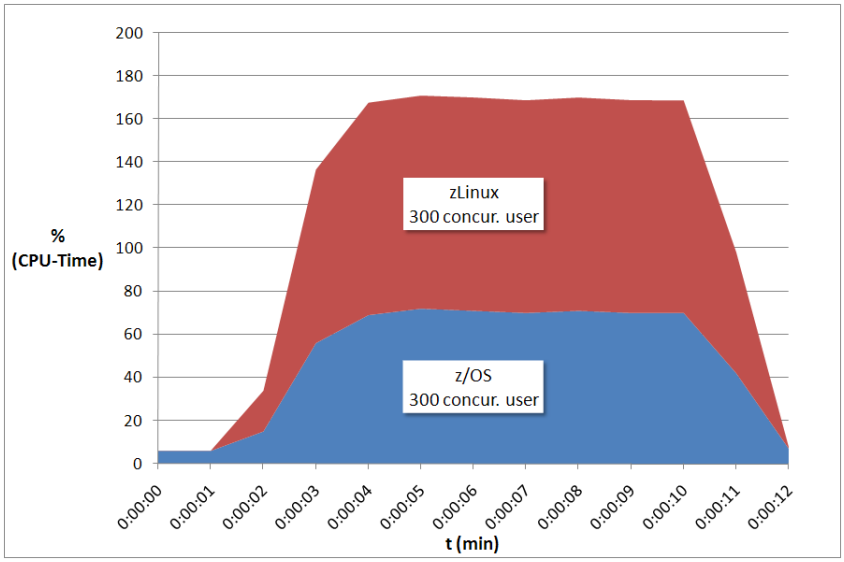


Abbildung 6.23: Prozessorauslastung - 300 gleichzeitige Benutzer - verteiltes Szenario.

Im Vergleich zu 200 (10min 5s) bzw. 250 (10min 6s) gleichzeitigen Benutzern verlängert sich

die Laufzeit des Tests auf 10min 59s. Die Leistungsfähigkeit des Prozessors der zLinux-LPAR ist erschöpft.

Diskussion der bisher gemachten Beobachtungen

Die verteilte Architektur mit zwei Prozessoren ist belastbarer als die integrierte Konfiguration mit einem Prozessor. Es gelingt der verteilten Konfiguration dabei nicht, beide Prozessoren voll zu nutzen. Sobald die zLinux-Partition ihre Saturierung erreicht hat, steigt auch die Belastung der Datenbankpartition nicht weiter an. Der Anwendungsserver kann nicht mehr Anfragen an die Datenbank stellen, als der Prozessor der zLinux-Partition zulässt (vgl. Abbildung 6.19, 6.21 und 6.23). Somit kann auch die Auslastung des Prozessors in der Datenbankpartition nicht mehr steigen, wenn der Prozessor der zLinux-Partition mit dem Anwendungsserver saturiert ist.

Das integrierte Szenario ist mit 150 gleichzeitigen Benutzern saturiert (vgl. Abbildung 6.16). Die verteilte Konfiguration schafft so betrachtet 33% mehr Belastung vor der Saturierung des Prozessors, denn sie ist mit 200 Benutzern voll ausgelastet. Die verteilte Konfiguration hat aber auch die doppelten Prozessorressourcen zur Verfügung, was in einer nachfolgenden, zweiten Testreihe geändert wird, um einen gültigen Vergleich der beiden Szenarien vornehmen zu können.

Die verteilte Installation erfährt durch dieselbe Last eine größere Prozessorbelastung, als die integrierte Installation. Vergleicht man Abbildung 6.14 mit Abbildung 6.15 sieht man, dass die verteilte Konfiguration ca 20% mehr CPU-Zeit als die integrierte Konfiguration für die Behandlung derselben Requests verbraucht. Die Ursache hierfür liegt in der Verarbeitung der Anfragen an die verteilt installierte Datenbank.

Bei einer entfernten Anfrage an die Datenbank werden die Anfrage- und Rückgabeparameter über das Netzwerk verschickt. Nachdem eine Verbindung zur Datenbank über ein DataSource-Objekt hergestellt wurde, wird die Anfrage gestellt. Dabei wird die Anfrage in ein sendefähiges Format umgewandelt. Diesen Vorgang bezeichnet man als *Marshalling*. Auf Datenbankseite wird die Anfrage wieder aus den Netzwerkpaketen extrahiert und in ein Format umgewandelt, das die Datenbank für eine Anfrage erwartet. Diesen Vorgang bezeichnet man als *Unmarshalling*.

Für die Rückrichtung wiederholt sich der Prozess. Das Ergebnis des gestellten Queries wird wieder in das Sendeformat umgewandelt, in Netzwerkpakete verpackt und auf Seiten des Anwendungsservers wieder aus den Netzwerkpaketen extrahiert.

Bei diesem Vorgang werden die Anfrageparameter und das Ergebnis der Anfrage *serialisiert*, d.h. aus den Objekten im Speicher wird ein Bytestrom erstellt. Dieser Vorgang verbraucht CPU-Zeit. Und dies löst den Mehrverbrauch an CPU-Zeit im verteilten Szenario aus.

Abbildung 6.24 veranschaulicht den beschrieben Vorgang in vereinfachter Form:

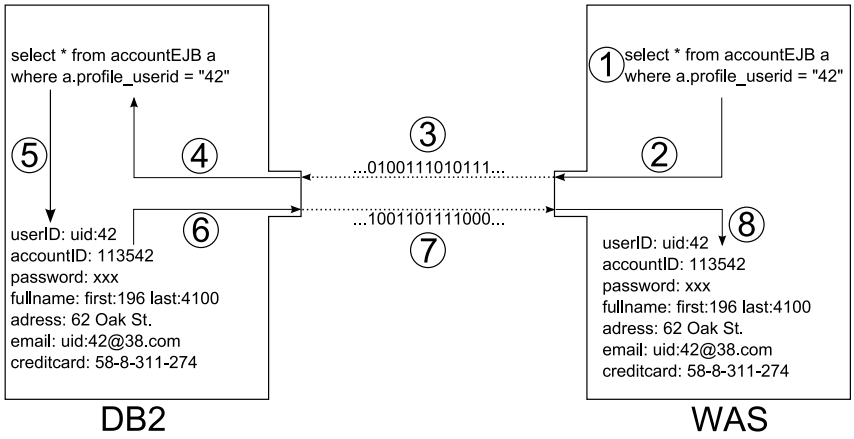


Abbildung 6.24: Ablauf einer entfernten Datenbankanfrage (vereinfacht).

1. Die Anfrage wird im Anwendungsserver erstellt und abgeschickt.
2. Die Anfrage wird für den Versand über das Netzwerk serialisiert.
3. In Form eines Bytestroms durchläuft die Anfrage das Netzwerk.
4. Im Adressraum der Datenbank wird aus dem Bytestrom ein Objekt im Speicher (Deserialisierung).
5. Die Anfrage wird ausgewertet und die Ergebnismenge im lokalen Speicher abgelegt.
6. Die Ergebnismenge wird zu einem Bytestrom serialisiert.
7. Der serialisierte Bytestrom durchläuft das Netzwerk zurück zum Anwendungsserver.
8. Im Anwendungsserver wird die Ergebnismenge deserialisiert und als Objekt im Speicher zur weiteren Verarbeitung durch die Anwendung bereitgestellt.

Die Schritte 2, 4, 6 und 8 sind hierbei jene, welche die Mehrbelastung im verteilten Szenario verursachen. Im integrierten Szenario sind diese Schritte nicht notwendig, da die Objekte direkt im Speicher kopiert werden können und nicht serialisiert werden müssen. Eine JDBC-Typ-2-Verbindung ist also ressourcenschonender, als eine JDBC-Typ-4-Verbindung.

Im Folgenden wird betrachtet, aus welchen Programmanteilen sich die CPU-Zeit zusammensetzt.

6.3.1 Verteilung auf die Adressräume/Reportklassen im integrierten Szenario

Bei oben aufgezeichneten Testläufen wurden weitere Metriken auf beiden Systemen erfasst:

- **% eappl by WLM report class:** Diese Metrik gibt an, wieviel CPU-Zeit prozentual von welcher, in WLM konfigurierten, Report-Klasse (vgl. S. 29) im Messintervall verbraucht wurde. Dabei enthält der Wert die Summe aus TCB- und SRB-Zeit **inklusive** Enclaven.

Es ist wichtig zu beachten, dass die CPU-Zeit in Enclaven hierbei mit eingeschlossen wird. Man erinnere sich: In den Enclaven wird adressraumübergreifend der Request an den WebSphere Application Server bearbeitet. Die Enclaven des WebSphere-Servers beinhalten auch die CPU-Zeit, die in den Adressräumen der Datenbank verbraucht wird. Die Datenbank wird im folgenden Diagramm kaum separat auftauchen, da in den Reportklassen der Datenbank nur sehr wenig CPU-Zeit verbraucht wurde, die nicht den Anfragen aus dem Anwendungsserver zuzurechnen ist.

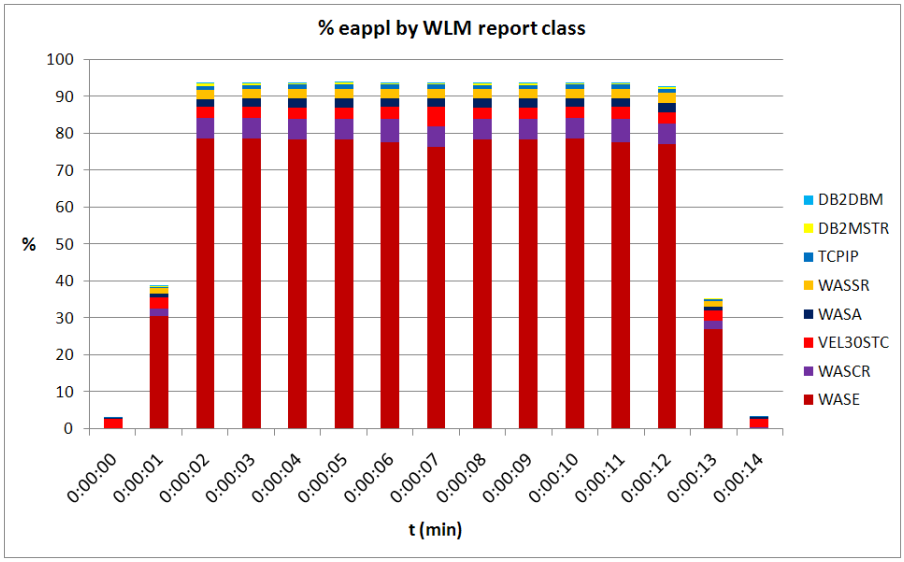


Abbildung 6.25: Verteilung der Last auf Reportklassen im integrierten Szenario bei 150 gleichzeitigen Benutzern.

In ersten Versuchsmessungen wurde festgestellt, dass die WLM-Konfiguration nicht korrekt eingerichtet war und diese wurde dann nach den Empfehlungen in [uAD+07] konfiguriert. Dabei wurden u.a. folgende Reportklassen festgelegt:

- **WASE:** Die Default-Reportklasse für die Enclaven des WebSphere Application Servers. In den Messungen wird die Default-Klasse für alle Enclaven des WAS verwendet. Möchte man verschiedene Requests verschieden klassifizieren, so ist dies über sogenanntes *Transaction Class Mapping* möglich. Hierbei werden Requests anhand des angesprochenen, virtuellen Hosts, Portnummern oder URIs unterschiedlichen Klassen zugeordnet (vgl. [uAD+07]). Da auf dem für diese Arbeit verwendeten Server aber nur eine Anwendung (Trade V6.1) läuft, konnte für alle Requests die Default-Klasse verwendet werden.
- **WASCR:** Die Reportklasse für die Control Region des WAS.
- **VEL30STC:** Die Default-Klasse für *System Started Tasks*. Hierunter fallen z.B. der RMF Data Gatherer oder der Workload Manager.
- **WASA:** Die Reportklasse für die Control Region Adjunct des WAS.

- **WASSR:** Unter diese Reportklasse läuft die Servant Region des WAS. Last, die unter dieser Klasse abgerechnet wird, wird nicht direkt von den ankommenden Requests verursacht, sondern z.B. vom Garbage Collector der JVM.
- **TCPIP:** Dies ist die Reportklasse für den Job, der unter z/OS für die Verarbeitung IP-basierter Kommunikation zuständig ist.
- **DB2MSTR und DB2DBM:** Zwei Reportklassen für DB2-Adressräume.

Den größten Anteil an der CPU-Zeit haben in Abbildung 6.25 mit knapp 80% die Enclaven des WebSphere Application Servers. Weitere 10% der CPU-Zeit gehen zu Lasten der drei Regions, aus denen sich eine Instanz des WebSphere Application Servers auf z/OS zusammensetzt. Allerdings wird diese Last nur indirekt durch die ankommenden Requests verursacht und beinhaltet z.B. durch den Garbage Collector oder den JIT-Compiler verursachte CPU-Zeit.

Erfasst man obige Metrik nicht für Reportklassen sondern Adressräume (Jobs), erhält man folgendes Bild:

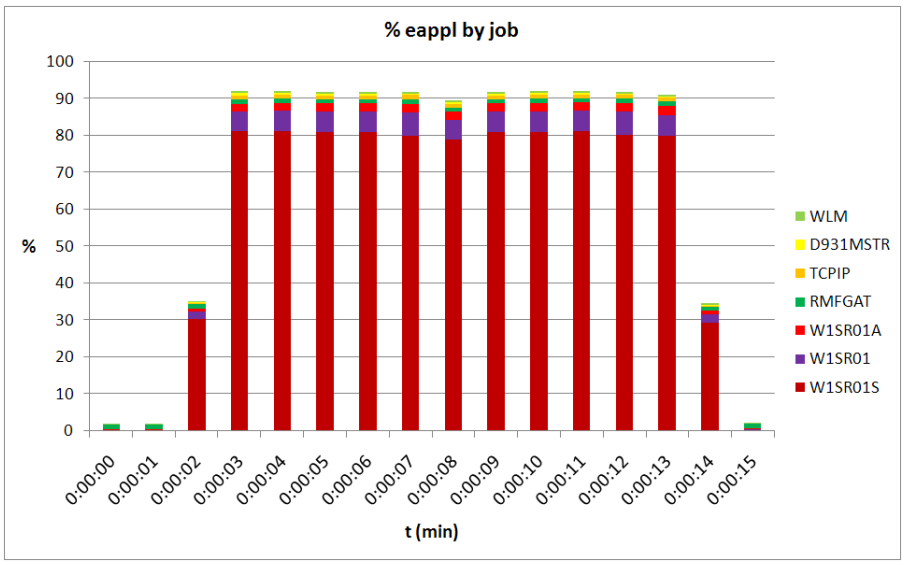


Abbildung 6.26: Verteilung der Last auf Adressräume im integrierten Szenario bei 150 gleichzeitigen Benutzern

Hier tauchen die Adressräume WLM und RMFGAT auf, die in der Betrachtung über Reportklassen in der Klasse VEL30STC abgerechnet wurden. Ansonsten zeigt sich auch hier, dass der Großteil der Last in den Enclaven des Anwendungsservers verursacht wird.

6.3.2 Verteilung auf die Adressräume/Reportklassen im verteilten Szenario

Für die verteilte Konfiguration ändert sich jetzt das Bild auf z/OS-Seite. Es tauchen neue Reportklassen auf:

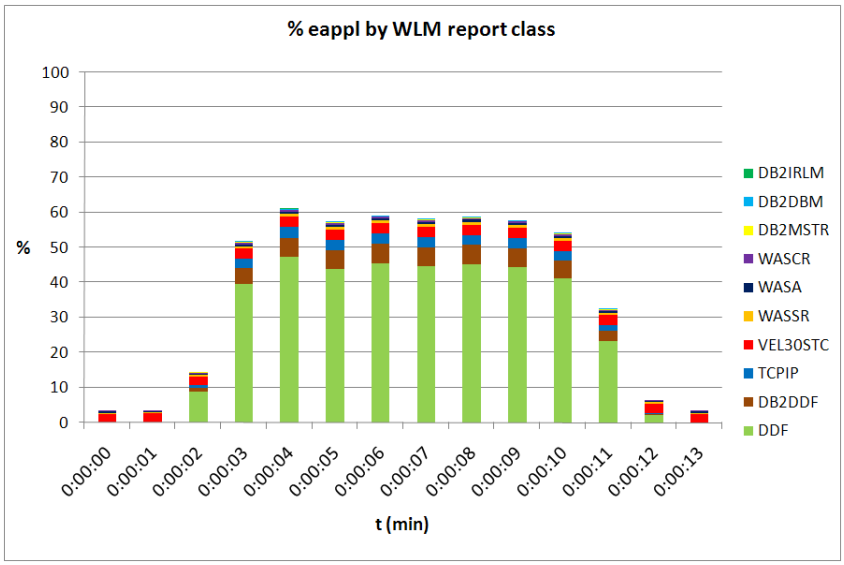


Abbildung 6.27: Verteilung der Last auf z/OS-Reportklassen im verteilten Szenario bei 150 gleichzeitigen Benutzern.

Entfernte Datenbankanfragen werden von DB2 ebenfalls in separaten Enclaven behandelt. Folgende Reportklassen sind in Abbildung 6.27 neu hinzugekommen:

- **DDF:** Die Reportklasse für die Distributed Data Facility, dem DB2-Kommunikationsendpunkt für entfernte Datenbankanfragen. Hier wird eine Enclave für ankommende Transaktionen erstellt und alle CPU-Zeit, die die Transaktion verbraucht, wird dieser Enclave zugerechnet. Die Transaktion geht aber auch hier über mehrere DB2-Adressräume.
- **DB2DDF:** Diese Reportklasse beinhaltet CPU-Zeit, die vom DDF-Adressraum verbraucht wird, aber nicht direkt einer Transaktion zuzurechnen ist.
- **DB2MSTR, DB2DBM und DB2IRLM:** Die restlichen Reportklassen für die drei anderen Adressräume, aus denen sich eine DB2-Instanz auf z/OS zusammensetzt (vgl. S.50).

Den Großteil der CPU-Zeit auf z/OS-Seite verbrauchen im verteilten Szenario die DB2-Enclaven, die durch die entfernten Datenbankanfragen entstehen. Man sieht hier auch, dass sich der Anteil der Netzwerkkommunikation in der Reportklasse TCPIP im Vergleich zum integrierten Szenario leicht erhöht. Das ist zu erwarten, denn wo im integrierten Szenario ein HTTP-Request ankommt, findet im verteilten Szenario mehr Netzwerkkommunikation statt. Für einen HTTP-Request an den Anwendungsserver werden mehrere Anfragen an die Datenbank gestellt.

Für die zLinux-Partition wird eine entsprechende Metrik für Adressräume erfasst:

- **% cpu time total by process:** Gibt die CPU-Zeit in User-, Nice und Kernel-Modus aufgeschlüsselt nach Prozessen an.

Die Daten werden hier aber vom Linux Data Gatherer fehlerhaft akkumuliert und teilweise erhält man für den relevanten Java-Prozess des WebSphere Application Server keine oder sogar fehlerhafte, sprich negative, Werte.

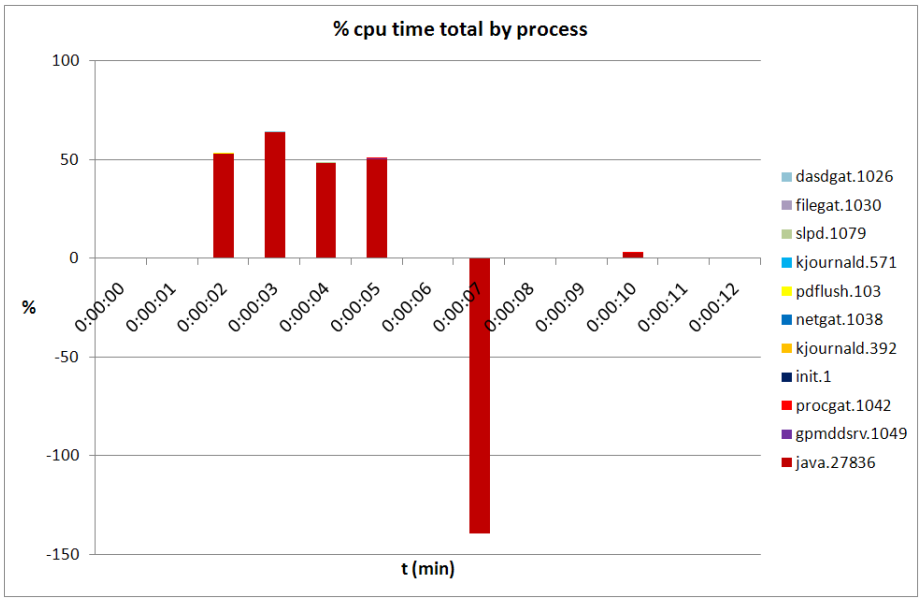


Abbildung 6.28: CPU-Zeit pro Prozess auf zLinux im verteilten Szenario bei 150 gleichzeitigen Benutzern (fehlerhaft!).

Über diese Daten kann keine Auswertung stattfinden. Der Linux Data Gatherer wird derzeit nicht mehr von IBM betreut und eine Ursache für dieses Problem konnte nicht gefunden werden.

Um sicherzustellen, dass der Großteil der totalen CPU-Auslastung innerhalb der zLinux-Partition vom Anwendungsserver verursacht wird, werden einmalig die CPU-Zeiten des java-Prozesses, der eine WAS-Instanz bildet, manuell auf Basis der Ausgabe des Linux-Programms *top* mitprotokolliert und akkumuliert. Dabei wurde für top ein Aktualisierungsintervall von 2s festgelegt und die erhaltenen Daten minutenweise gemittelt. Das Ergebnis ist in folgendem Diagramm zu sehen.

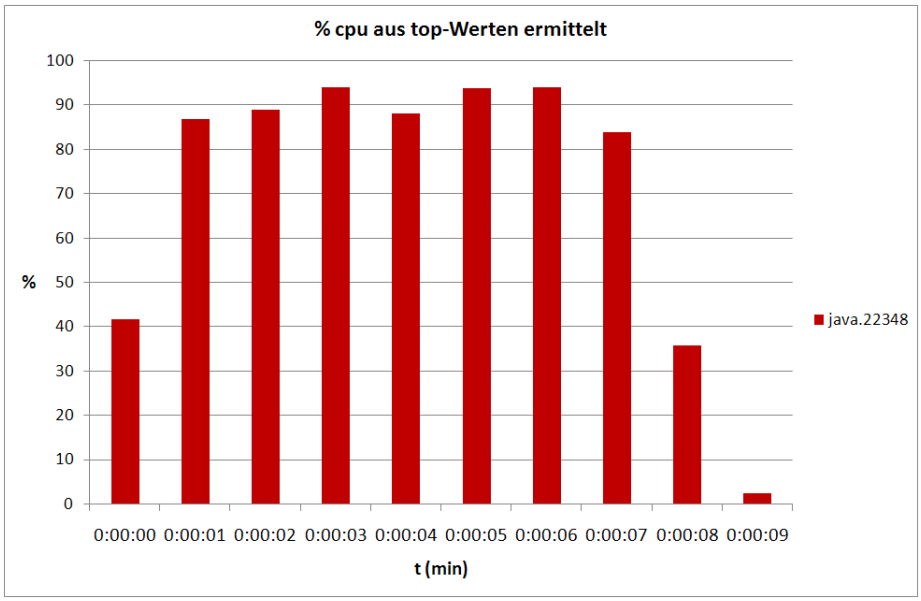


Abbildung 6.29: Manuell erfasste CPU-Zeiten für den Java-Prozess auf zLinux und 150 gleichzeitige Benutzer.

Vergleicht man Abbildung 6.29 mit Abbildung 6.17, S. 79, so hat man Gewissheit, dass die totale CPU-Auslastung innerhalb der zLinux-Partition im verteilten Szenario zum größten Teil vom darauf ablaufenden WebSphere Application Server verursacht wird. Die ermittelten Werte sind allerdings mit Vorsicht zu bewerten, da sie manuell erfasst wurden.

Durch die bisherige Konfiguration der Messumgebung findet ein „unfairer Vergleich“statt. Im verteilten Szenario wird die Last auf die beiden Prozessoren der beiden LPARs der zLinux- und z/OS-Partition verteilt. Der WebSphere-Server und die Datenbank haben je einen Prozessor für sich. Im integrierten Szenario ist bisher nur ein Prozessor für die Last von Anwendungsservern und Datenbank in der z/OS-Partition konfiguriert. Um den Vergleich aussagekräftiger zu machen, werden die durchgeführten Testläufe wiederholt. In der zweiten Testreihe wird der z/OS-Partition ein zweiter Prozessor dediziert zugewiesen. Erst nach dieser Anpassung der Testumgebung ist ein aussagekräftiger Vergleich möglich und beiden Szenarien stehen nun diesselben Prozessorressourcen zur Verfügung.

6.3.3 Erweiterung des integrierten Szenarios auf zwei Prozessoren

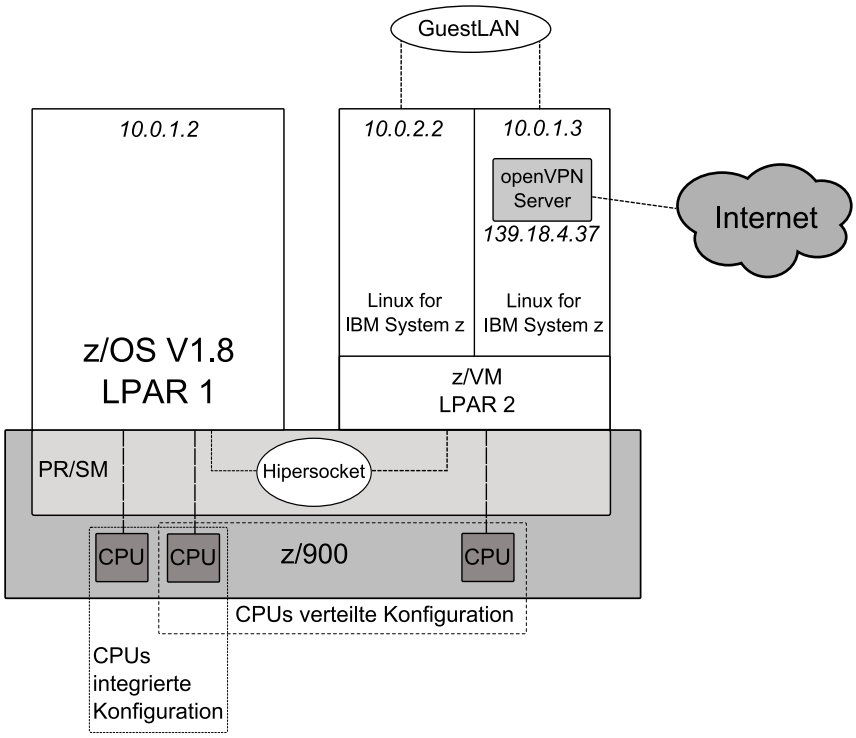


Abbildung 6.30: Überblick über die verwendete LPAR-Architektur der zweiten Testreihe.

Die folgenden Diagramme zeigen die Prozessorauslastung des integrierten Szenarios mit zwei Prozessoren. Die erfasste Metrik ist wieder *% total utilization*. Für mehrere Prozessoren wird in dieser Metrik die durchschnittliche Auslastung **pro** Prozessor angezeigt, d.h. eine Auslastung von 50% sagt aus, dass jeder Prozessor zu je 50% ausgelastet war. Die so erfasste CPU-Zeit ist für einen Vergleich mit der verteilten Konfiguration also zu verdoppeln, um auf die Summe der Auslastung aller beteiligten Prozessoren zu kommen. Für die Darstellung in den Diagrammen wird der erfasste Wert verdoppelt. Die maximal mögliche Auslastung des Systems liegt, wie im verteilten Szenario, bei 200%.

Testlauf mit 50 gleichzeitigen Benutzern und zwei Prozessoren für die z/OS-LPAR

Diagramm 6.31 zeigt die CPU-Auslastung in einem ersten Testlauf mit 50 gleichzeitigen Benutzern nach der Erweiterung der z/OS-LPAR auf zwei dedizierte Prozessoren. Über die x-Achse ist wieder die Zeit in Minuten und über die y-Achse die Auslastung der Prozessoren in % abgetragen.

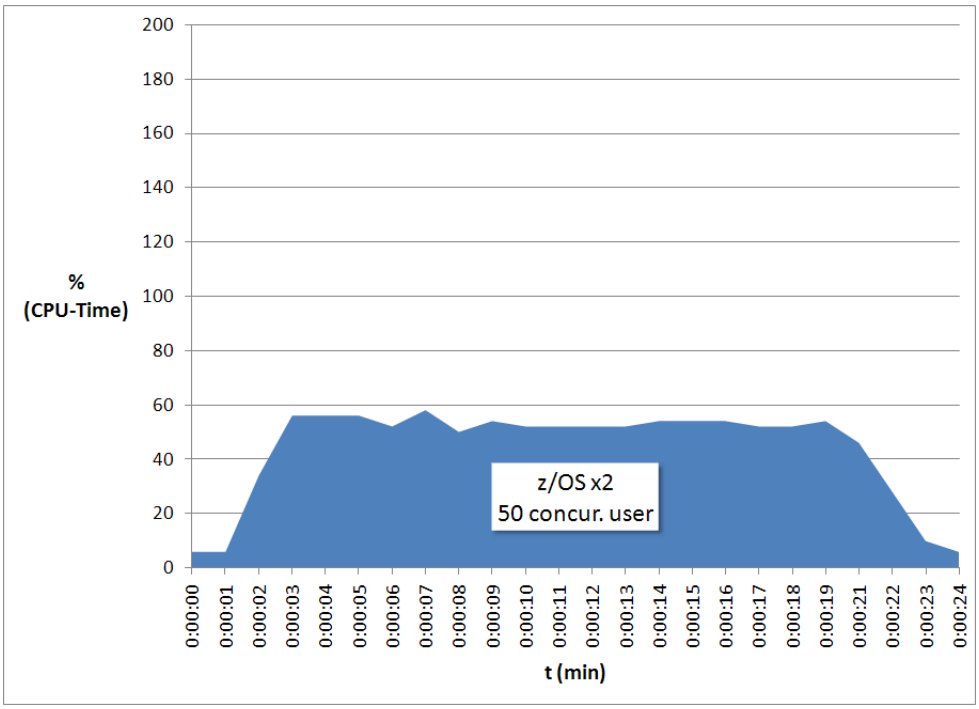


Abbildung 6.31: Prozessorauslastung zweier Prozessoren - 50 gleichzeitige Benutzer - integriertes Szenario.

Die Auslastung der beiden Prozessoren liegt bei jeweils knapp 27%, was eine Gesamtauslastung der Prozessorressourcen zu ungefähr 55% ergibt. Im Vergleich zur Gesamtauslastung der verteilten Konfiguration in Abbildung 6.13, S. 76, ergibt sich wieder eine Differenz von ca. 6 - 10% zugunsten der integrierten Konfiguration.

Testlauf mit 100 gleichzeitigen Benutzern und zwei Prozessoren für die z/OS-LPAR

Wie in der ersten Testreihe wird auch hier die Anzahl der gleichzeitigen Benutzer pro Testlauf um 50 erhöht. Diagramm 6.32 zeigt die CPU-Auslastung des integrierten Szenarios für 100 gleichzeitige Benutzer und zwei dedizierten Prozessoren.

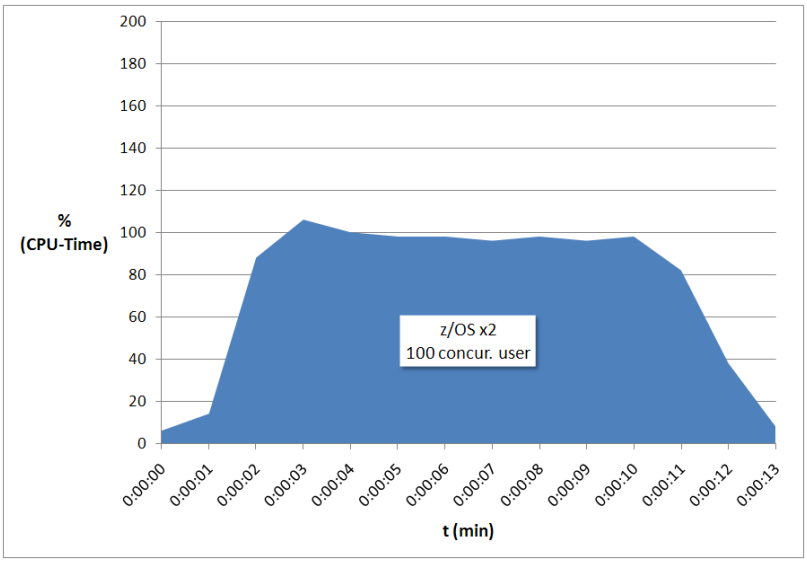


Abbildung 6.32: Prozessorauslastung zweier Prozessoren - 100 gleichzeitige Benutzer - integriertes Szenario.

Die integrierte Konfiguration skaliert sehr gut. Im Vergleich zum verteilten Szenario werden die Prozessorressourcen um ca. 20% weniger belastet (vgl. Abbildung 6.15, S. 78).

Testlauf mit 150 gleichzeitigen Benutzern und zwei Prozessoren für die z/OS-LPAR

In Diagramm 6.33 ist der zeitliche Verlauf der CPU-Auslastung für einen Testlauf mit 150 gleichzeitigen Benutzern dargestellt.

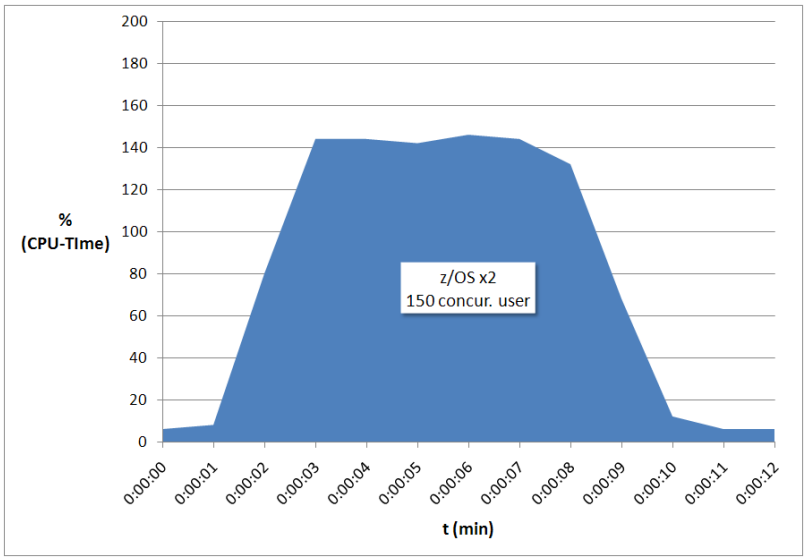


Abbildung 6.33: Prozessorauslastung zweier Prozessoren - 150 gleichzeitige Benutzer - integriertes Szenario.

An dieser Stelle war das integrierte Szenario mit einem dedizierten Prozessor für die z/OS-LPAR an seiner Belastungsgrenze angelangt. Die Auslastung zweier Prozessoren erreicht hier in der Summe ca. 140%. Das ergibt eine Differenz zur Gesamtauslastung im verteilten Szenario von ca. 18% (vgl. Abbildung 6.17, S. 79).

Testlauf mit 200 gleichzeitigen Benutzern und zwei Prozessoren für die z/OS-LPAR

Für die in Diagramm 6.34 dargestellte Messung wird die Anzahl der gleichzeitigen Benutzer auf 200 erhöht.

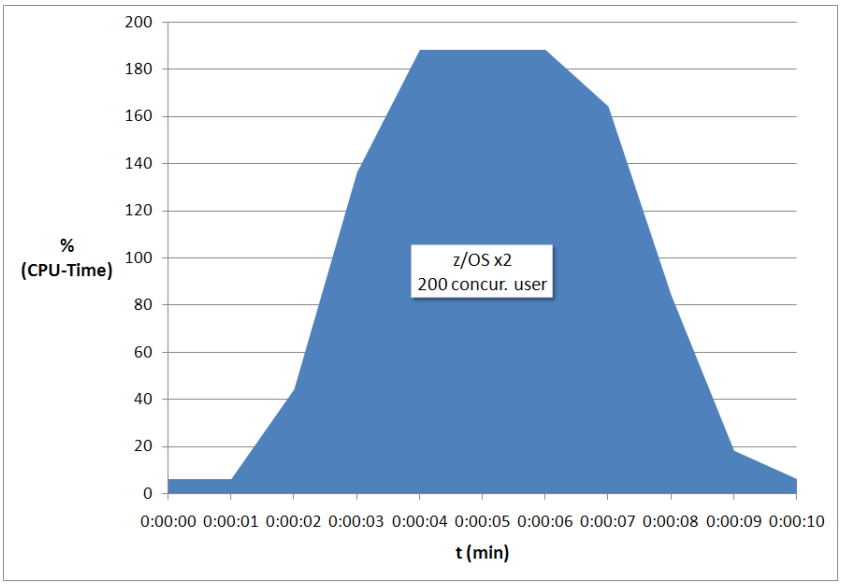


Abbildung 6.34: Prozessorauslastung zweier Prozessoren - 200 gleichzeitige Benutzer - integriertes Szenario.

Im Gegensatz zum verteilten Szenario ist das integrierte Szenario mit zwei Prozessoren hier noch nicht saturiert. Abbildung 6.19, S. 81 zeigt, dass die zLinux-Partition vollständig ausgelastet ist. Dadurch übersteigt die Gesamtauslastung der integrierten Konfiguration mit zwei Prozessoren hier die Gesamtauslastung der verteilten Konfiguration und es können keine Aussagen mehr über einen möglichen Overhead durch die JDBC-Konfiguration getroffen werden.

Testlauf mit 250 gleichzeitigen Benutzern und zwei Prozessoren für die z/OS-LPAR

In diesem vorletzten Testlauf der Testreihe greifen 250 Benutzer gleichzeitig auf den Trade-Benchmark zu. Den dabei erfassten Verlauf der CPU-Zeit zeigt Diagramm 6.35.

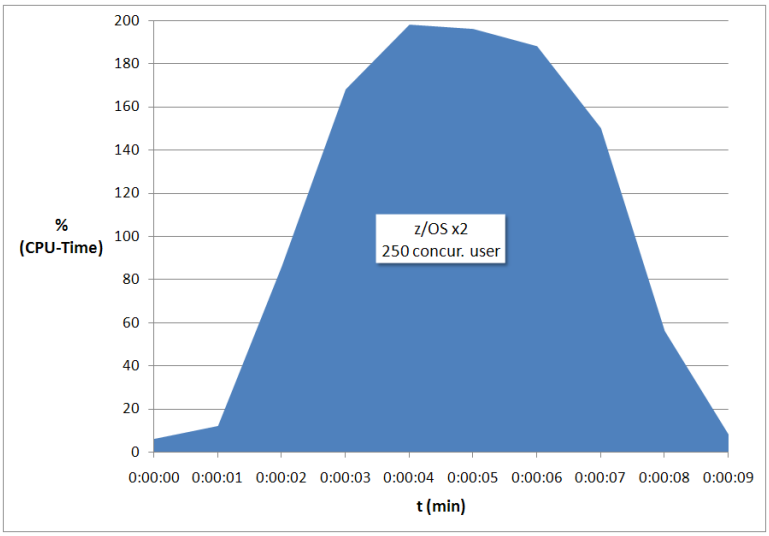


Abbildung 6.35: Prozessorauslastung zweier Prozessoren - 250 gleichzeitige Benutzer - integriertes Szenario.

Mit 250 Benutzern ist das integrierte Szenario noch nicht voll ausgelastet. Die Konsolidierung von Datenbank und Anwendungsserver führt zu einer Leistungssteigerung und einer besseren Nutzung der gegebenen Prozessorressourcen.

Testlauf mit 300 gleichzeitigen Benutzern und zwei Prozessoren für die z/OS-LPAR

Diagramm 6.36 zeigt die CPU-Auslastung im letzten Testlauf der zweiten Testreihe über die Zeit mit 300 gleichzeitigen Benutzern im integrierten Szenario mit zwei dedizierten Prozessoren.

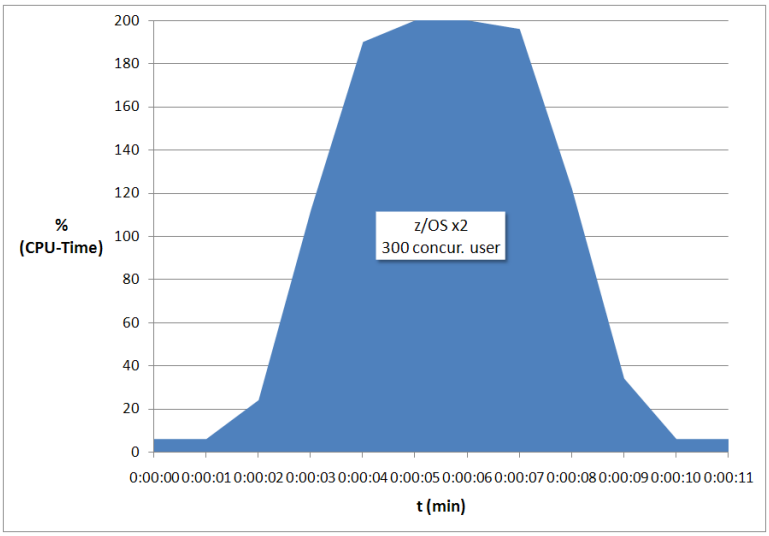


Abbildung 6.36: Prozessorauslastung zweier Prozessoren - 300 gleichzeitige Benutzer - integriertes Szenario.

Mit 300 gleichzeitigen Benutzern ist die integrierte Architektur mit zwei Prozessoren voll ausgelastet und erreicht ihre Saturierung. Das integrierte Szenario mit zwei Prozessoren bewältigt somit im Vergleich zum verteilten Szenario 50% mehr Belastung. Das verteilte Szenario ist mit 200 gleichzeitigen Benutzern voll ausgelastet.

6.3.4 Zusammenfassung der Untersuchung über die Prozessorauslastung

Als Ergebnis der Untersuchung der Prozessorauslastung der integrierten und verteilten Konfiguration haben sich mehrere interessante Punkte herausgestellt.

Eine integrierte Konfiguration mit einem Prozessor ist weniger belastbar als eine verteilte Konfiguration mit zwei Prozessoren. Allerdings hat sich schon hier gezeigt, dass die verteilte Konfiguration bei gleicher Belastung des Anwendungsservers in der Summe mehr CPU-Zeit verbraucht, als die integrierte Konfiguration. Hauptgrund hierfür ist die unterschiedliche Konfiguration des Datenbankzugriffs über einen JDBC-Treiber Typ 4 bzw. Typ 2.

Stellt man dem integrierten Szenario dieselben Prozessorressourcen wie der verteilten Konfiguration zur Verfügung, zeigen sich zwei weitere Vorteile der integrierten Konfiguration. Eine konsolidierte Lösung auf System z ist zu 50% leistungsfähiger als die verteilte Installation. Desweiteren ist nur die integrierte Konfiguration in der Lage, die gegebenen Prozessorressourcen voll zu nutzen (vgl. Abbildung 6.23, S. 83, mit Abbildung 6.36), was der Grund für die höhere Leistung ist.

Eine Datenbankanbindung über einen JDBC-Treiber Typ 2 verbraucht weniger CPU-Zeit als eine Anbindung über einen JDBC-Treiber Typ 4. Konfiguriert man im integrierten Szenario einen JDBC-Treiber Typ 4, steigt der Verbrauch an CPU-Zeit stark an und übersteigt sogar teilweise den CPU-Verbrauch des verteilten Szenarios, wie die Abbildungen 6.37 und 6.38 zeigen:

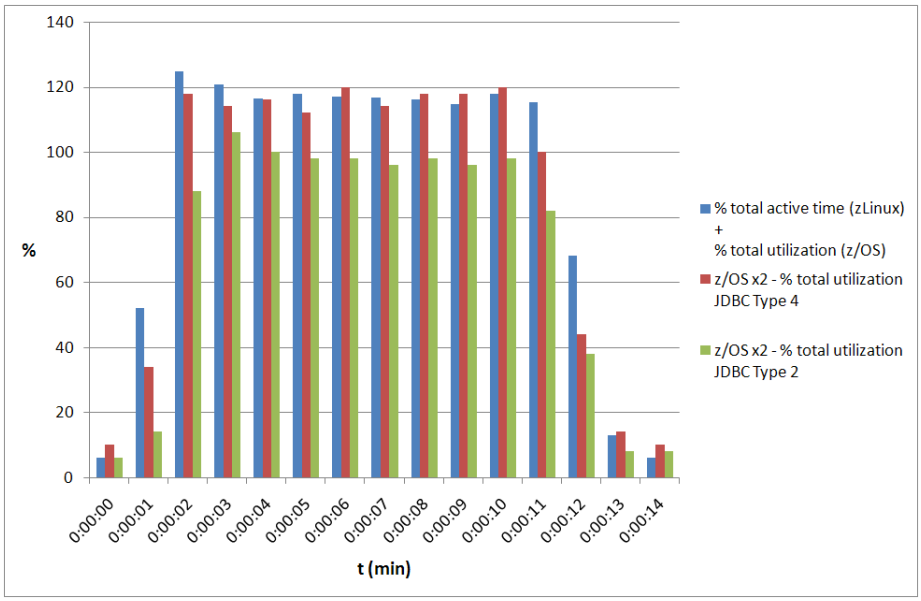


Abbildung 6.37: CPU-Verbrauch im verteilten und integrierten Szenario mit 100 gleichzeitigen Benutzern und JDBC-Typ-2- bzw. JDBC-Typ-4-Konfiguration

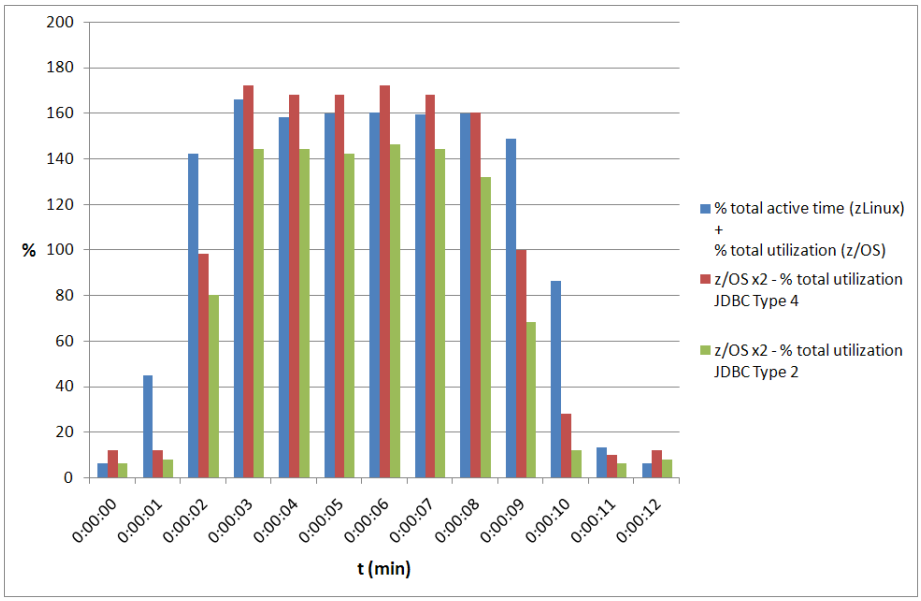


Abbildung 6.38: CPU-Verbrauch im verteilten und integrierten Szenario mit 150 gleichzeitigen Benutzern und JDBC-Typ-2- bzw. JDBC-Typ-4-Konfiguration

Die beiden vorhergehenden Abbildungen stellen den Gesamtverbrauch an CPU-Zeit für drei Szenarien dar:

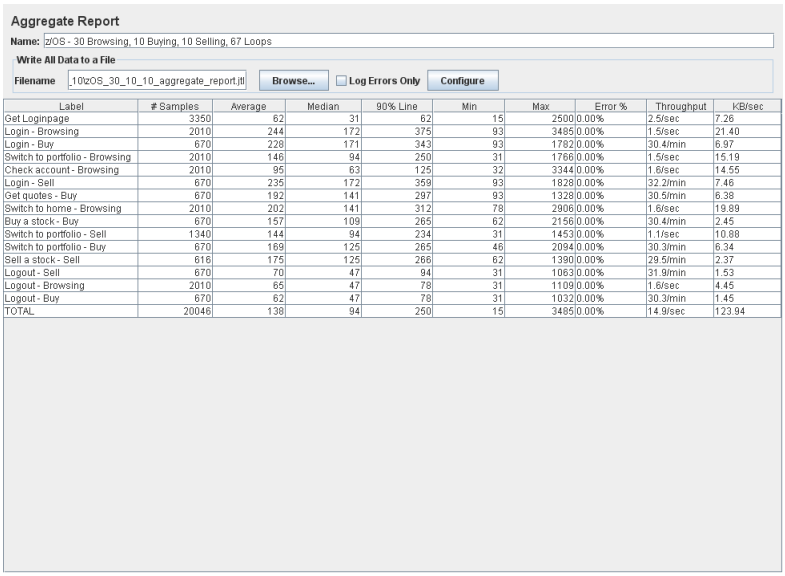
- **% total active time (zLinux) + % total utilization (z/OS):** Der gesamte Verbrauch an CPU-Zeit im verteilten Szenario mit Datenbankanbindung über einen JDBC-Treiber Typ 4.
- **z/OS x2 - % total utilization JDBC Type 4:** Das integrierte Szenario ist für diesen Testlauf mit zwei Prozessoren ausgestattet. Die Datenbankanbindung erfolgt über einen JDBC-Treiber Typ 4.
- **z/OS x2 - % total utilization JDBC Type 2:** Das integrierte Szenario ist für diesen Testlauf mit zwei Prozessoren ausgestattet. Die Datenbankanbindung erfolgt über einen JDBC-Treiber Typ 2.

Diese Auswertung zeigt, dass eine Datenbankkonfiguration über einen JDBC-Treiber Typ 4 ca. 20% mehr CPU-Zeit benötigt, als eine Datenbankanbindung über einen JDBC-Treiber Typ 2. Das integrierte Szenario verbraucht bei Konfiguration eines JDBC-Treibers Typ 4 teilweise sogar mehr CPU-Ressourcen, als das verteilte Szenario bei gleicher Konfiguration und Prozessorausstattung (vgl. Abbildung 6.38).

In einer nächsten Untersuchung werden die über jMeter erfassten Daten miteinander verglichen.

6.4 Auswertung der jMeter-Reports

In jMeter lassen sich für die vorgenommenen Testläufe in einem aggregierten Report (engl. *aggregate report*) Informationen zu jedem Request-Sampler anzeigen. Abbildung 6.39 zeigt den Screenshots eines solchen Reports.



Label	# Samples	Average	Median	90% Line	Min	Max	Error %	Throughput	KB/sec
Get Loginpage	3350	62	31	62	15	2500	0.00%	2.5/sec	7.26
Login - Browsing	2010	244	172	375	93	3485	0.00%	1.5/sec	21.40
Login - Buy	670	228	171	343	93	1782	0.00%	30.4/min	6.97
Switch to portfolio - Browsing	2010	146	94	250	31	1766	0.00%	1.5/sec	15.19
Check account - Browsing	2010	95	63	125	32	3344	0.00%	1.6/sec	14.55
Login - Sell	670	235	172	359	93	1828	0.00%	32.2/min	7.46
Get quotes - Buy	670	192	141	297	93	1328	0.00%	30.5/min	6.38
Switch to home - Browsing	2010	202	141	312	78	2906	0.00%	1.6/sec	19.89
Buy a stock - Buy	670	157	109	265	62	2156	0.00%	30.4/min	2.45
Switch to portfolio - Sell	1340	144	94	234	31	1453	0.00%	1.1/sec	10.88
Switch to portfolio - Buy	670	169	125	265	46	2094	0.00%	30.3/min	6.34
Sell a stock - Sell	616	175	125	266	62	1390	0.00%	29.5/min	2.37
Logout - Sell	670	70	47	94	31	1063	0.00%	31.9/min	1.53
Logout - Browsing	2010	65	47	78	31	1109	0.00%	1.6/sec	4.45
Logout - Buy	670	62	47	78	31	1032	0.00%	30.3/min	1.45
TOTAL	20046	138	94	250	15	3485	0.00%	14.9/sec	123.94

Abbildung 6.39: jMeter-Screenshot eines aggregierten Reports.

Für jeden Sampler wird eine Zeile mit folgenden Werten dargestellt:

- **Label:** Der Bezeichner des Samplers.
- **# Samples:** Die Anzahl der Ausführungen des Samplers.
- **Average:** Die durchschnittliche Antwortzeit in ms.
- **Median:** Der Median für die Antwortzeit in ms. Im Gegensatz zum Durchschnitt teilt der Median die Menge aller Antwortzeiten in zwei Hälften und ist so gegenüber Ausreißern robuster, sprich 50% der Requests liegen oberhalb dieser Zeit und 50% darunter.
- **90% Line:** Die maximale Zeit für 90% der vorgenommenen Anfragen dieses Samplers.
- **Min:** Das Minimum der für diesen Sampler erfassten Antwortzeiten.
- **Max:** Das Maximum der für diesen Sampler erfassten Antwortzeiten.
- **Error %:** Der Anteil der mit einem Fehler beantworteten Requests. Bei korrekter Konfiguration des Anwendungsservers kam es so gut wie nie zu fehlerhaft beantworteten Requests.
- **Throughput:** Der Durchsatz, gemessen in Requests pro Sekunde/Minute/Stunde.
- **kB/sec:** Der Durchsatz, gemessen in Kilobyte pro Sekunde.

Ein Teil der so erhaltenen Daten wird auf den folgenden Seiten analysiert.

6.4.1 Durchsatz

Im folgenden Diagramm ist die Durchsatzrate in Requests pro Sekunde in den verschiedenen Konfigurationen dargestellt. Die y-Achse gibt dabei die Durchsatzrate in Requests pro Sekunde an. Auf der x-Achse sind die sechs Testreihen mit der jeweiligen Benutzeranzahl abgetragen.

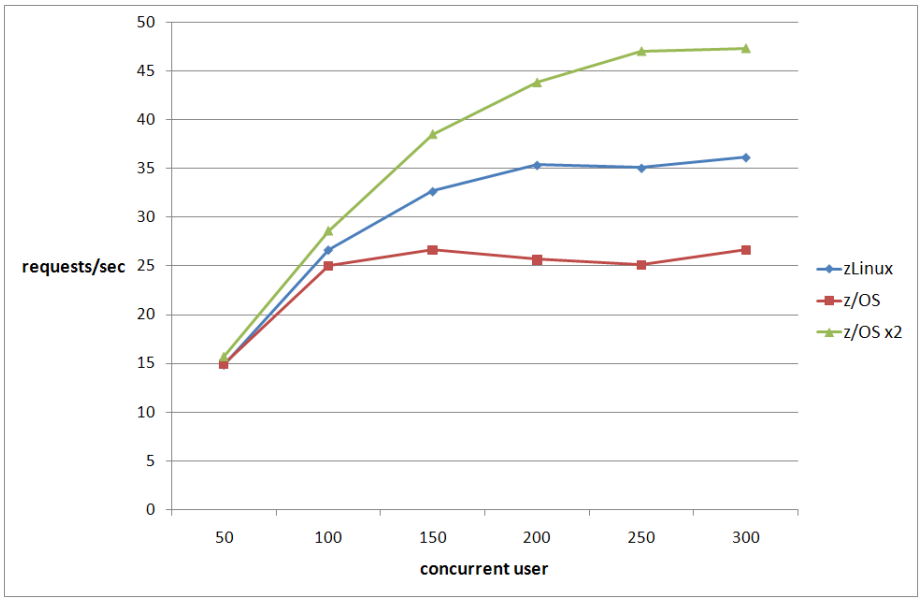


Abbildung 6.40: Durchsatz der jeweiligen Konfiguration bei steigender Zahl gleichzeitiger Benutzer.

- **zLinux:** Verteilte Konfiguration mit JDBC-Treiber Typ 4.
- **z/OS:** Integrierte Konfiguration mit JDBC-Treiber Typ 2 und einem dedizierten Prozessor.
- **z/OSx2:** Integrierte Konfiguration mit JDBC-Treiber Typ 2 und zwei dedizierten Prozessoren.

Der Durchsatz ist abhängig von der in jMeter konfigurierten Pause zwischen den einzelnen Requests innerhalb einer Threadgruppe (vgl. *Timer*, S. 68). Daher zeigen sich deutliche Unterschiede zwischen den Konfigurationen erst bei höherer Belastung durch mehr gleichzeitige Benutzer. Diese Darstellung ist ein weiterer Beleg für die unterschiedliche Belastbarkeit der verschiedenen Konfigurationen. Man sieht die unterschiedlichen Saturierungspunkte der Szenarien an den abfallenden Werten bei 150 und 200 gleichzeitigen Benutzern für das integrierte Szenario mit einem Prozessor bzw. das verteilte Szenario.

Interessanterweise steigt der Durchsatz für diese beiden Szenarien bei 300 Benutzern wieder leicht an. Die Ursache hierfür liegt in der Architektur der Messung. Für jede Messung gibt es eine sogenannte *Ramp-Up Period*. Diese, in Sekunden angegebene Zeitspanne, gibt an, in welcher Zeit die volle Anzahl paralleler Threads erstellt werden soll. Es wäre für den Rechner, auf dem der Lastgenerator ausgeführt wird, eine zu große Belastung 300 Threads gleichzeitig zu erzeugen und

dies würde die Messung verfälschen. Die Ramp-up Period wird in Abhängigkeit von der Gesamtthreadzahl eingestellt.. Das führt zu dem Effekt, dass bei 300 gleichzeitigen Benutzern diese 300 Threads langsamer aufgebaut werden, als die 250 Threads für 250 gleichzeitige Benutzer. Dies beeinflusst den Durchschnittswert des Durchsatzes auf oben ersichtliche Weise.

Die Ramp-Up Period hat allerdings keinen Einfluss auf folgenden Wert, der die durchschnittliche Antwortzeit pro Sampler betrachtet.

6.4.2 Durchschnittliche Antwortzeit

Die Antwortzeit ist für den Benutzer einer Anwendung von großer Relevanz und bestimmt seine Arbeitsgeschwindigkeit. Die zuvor gemachte Untersuchung über die Prozessorauslastung sind für die Nutzer einer Anwendung nur in zweiter Linie wichtig. Eine Überlastung der Prozessorressourcen führt fast immer zu einer Verschlechterung der Antwortzeiten. Warum nur „fast immer“ wird in folgender Untersuchung gezeigt.

In den folgenden drei Diagrammen sind die durchschnittlichen Antwortzeiten für alle neun Requestsampler der Testläufe aufgeführt. Jeweils für einen Testlauf mit 100, 200 und 300 gleichzeitigen Benutzern und die jeweiligen Konfigurationen.

- **zLinux:** Die verteilte Konfiguration mit JDBC-Treiber Typ 4.
- **z/OS x1:** Die integrierte Konfiguration mit JDBC-Treiber Typ 2 und einem Prozessor.
- **z/OS x2:** Die integrierte Konfiguration mit JDBC-Treiber Typ 2 und zwei Prozessoren.

Auf der y-Achse ist die durchschnittliche Antwortzeit in ms abgetragen. Auf der x-Achse werden die oben genannten Konfigurationen gruppiert dargestellt.

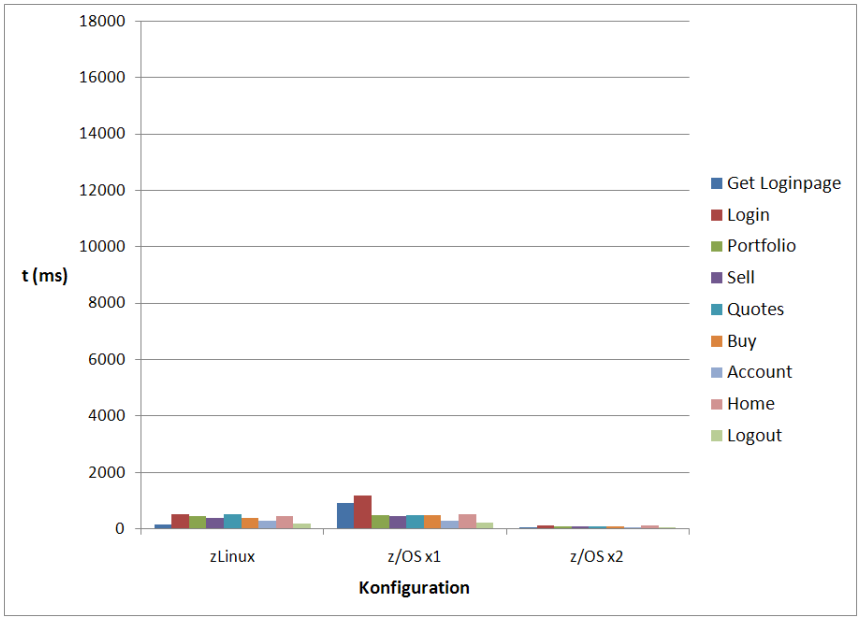


Abbildung 6.41: Durchschnittliche Antwortzeit bei 100 gleichzeitigen Benutzern.

Diese Last bewältigen alle drei Konfigurationen mit ausreichender Performanz. Das integrierte Szenario mit zwei Prozessoren sticht hier aber durch seine verhältnismäßig schnellen Antwortzeiten heraus.

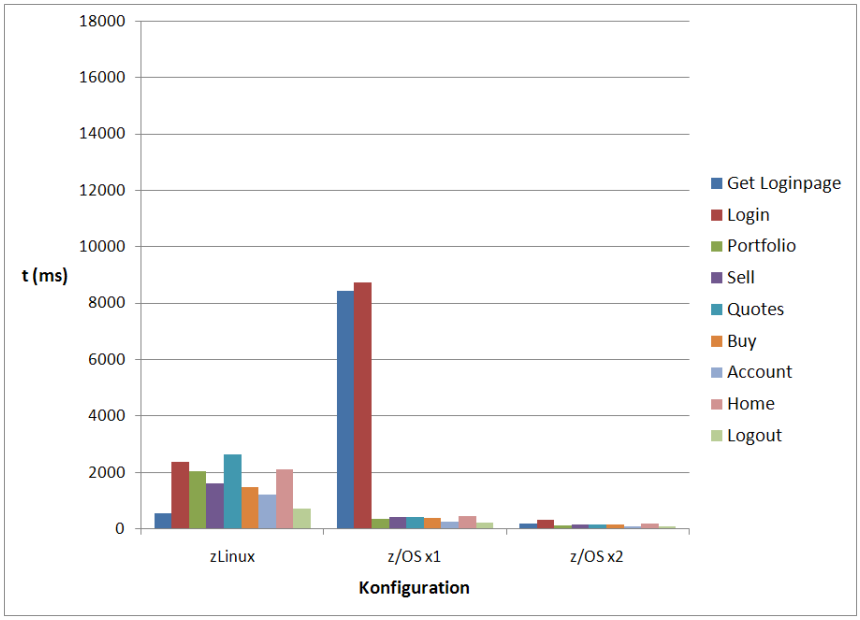


Abbildung 6.42: Durchschnittliche Antwortzeit bei 200 gleichzeitigen Benutzern.

Bei 200 gleichzeitigen Benutzern zeigen sich erste Überlastungserscheinungen im verteilten und im integrierten Szenario mit einem Prozessor. Das integrierte Szenario mit zwei Prozessoren behält sein gutes Antwortverhalten. Auffallend ist hierbei, dass sich das Antwortverhalten im verteilten Szenario gleichverteilt über alle Requestsampler verschlechtert. Im mit 200 Benutzern stark überlasteten integrierten Szenario mit einem Prozessor zeigen nur die Requestsampler *Get Loginpage* und *Login* eine Verschlechterung des Antwortverhaltens. Nach dem Login getätigte Requests werden in einer gleichbleibend guten Zeit vom Server behandelt.

Dieses Phänomen zeigt sich noch deutlicher in der folgenden Messung mit 300 gleichzeitigen Benutzern:

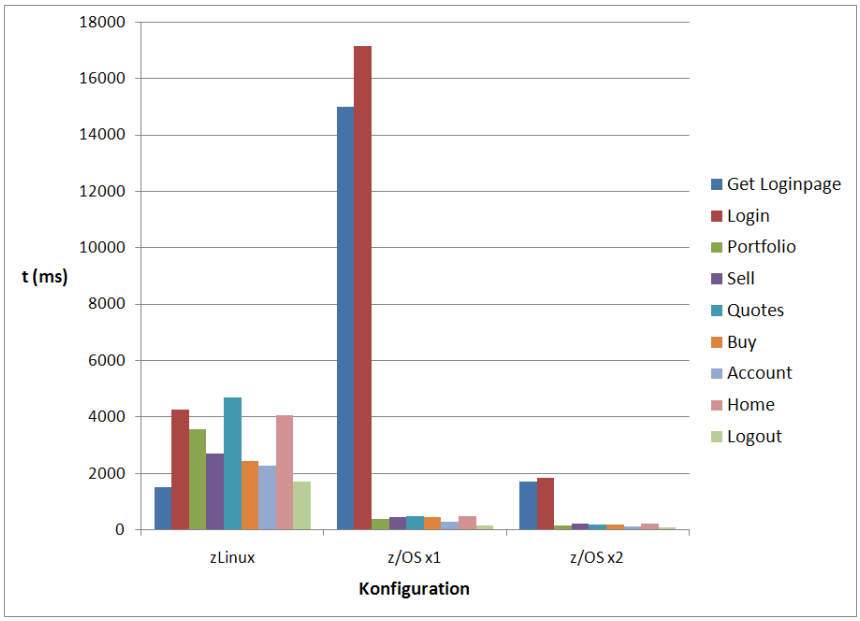


Abbildung 6.43: Durchschnittliche Antwortzeit bei 300 gleichzeitigen Benutzern.

Im verteilten Szenario verschlechtern sich die Antwortzeiten aller Requestsampler aufgrund der Überlastung gleichverteilt.

Das integrierte Szenario mit einem Prozessor ist hier völlig überlastet. Allerdings treten noch keine Timeouts auf und alle Requests werden beantwortet. Im integrierten Szenario mit zwei Prozessoren wird das Antwortverhalten marginal schlechter.

Wie oben bereits erwähnt ist für das integrierte Szenario auffallend, dass das Antwortverhalten auch bei Überlastung des Servers für einen Teil der Requestsampler konstant gut bleibt. Nur Requests, die vor dem Login gestellt werden, haben bei Überlastung der Serverressourcen ein verschlechtertes Antwortverhalten.

Ursache hierfür ist das in Kapitel 4 vorgestellte Workloadmanagement unter z/OS.

HTTP-Sessions und Workloadmanagement unter z/OS

Für die Sitzungsverwaltung verwendet der Trade6-Benchmark eine Variable `JSESSIONID`. Im Response des Requestsamplers *Login* wird diese mitgesendet und in Form eines Cookies auf dem Client abgelegt. Alternativ ließe sich die `JSESSIONID` auch über *URL-Rewriting* zwischen Client und Server austauschen. Dies lässt sich in der WebSphere-Konfiguration festlegen. Listing 6.1 zeigt eine Übersicht über den Response eines Login-Requests. Die `JSESSIONID` wurde zu Darstellungszwecken nach dem Separator (vgl. Tabelle 6.1) umgebrochen.

```
HTTP response headers:
HTTP/1.1 200 OK
Content-Type: text/html
Content-Language: en-US
Content-Length: 16122
Set-Cookie: JSESSIONID=0006JGSnsj9OonxVhSNQBnYAra:
                        C091997BF6B523880000009400000050A000102; Path=/
```

```
Date: Mon, 07 Jan 2008 12:56:14 GMT
Server: WebSphere Application Server/6.1
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Cache–Control: no–cache="set-cookie, set-cookie2"
```

Listing 6.1: Responseinhalt eines Login-Requests.

In einer WebSphere-Umgebung muss jeder Request, der Teil einer HTTP-Session ist, immer zur gleichen Serverinstanz in derselben JVM geleitet werden, in der die HTTP-Session erstellt wurde. Nur so ist eine konsistente Behandlung des Requests im Kontext der erstellten Sitzung gewährleistet. Dieses Verhalten wird als *Session Affinity* (Sitzungsaffinität) bezeichnet.

Zu diesem Zweck ist eine `JSESSIONID` mehrteilig aufgebaut und enthält folgende Informationen, die anhand eines Beispiels erläutert werden:

```
00006JGSnsj9OonxVhSNQBnYAra:C091997BF6B523880000009400000050A000102
```

Listing 6.2: Beispiel für eine JSESSIONID-Variable.

Das `JSESSIONID`-Cookie kann in vier Teile aufgespalten werden.

Inhalt	**Wert im Beispiel**
Cache-ID	0000
Session-ID	SHOQmBQ8EokAQtzl_HYdxlt
Seperator	:
Partition-ID	C091997BF6B523880000009400000050A000102

Tabelle 6.1: Bestandteile einer JSESSIONID

Neben der Session-ID, welche die erstellte Sitzung identifiziert, ist für die Session Affinity die Partition-ID von Bedeutung. Sie identifiziert die Anwendungsinstanz, sprich die Server-JVM, in der die Session erstellt wurde.

Über diese Identifizierung werden Anfragen, die zu einer bestehenden Sitzung gehören, direkt an die JVM weitergeleitet, in der die Sitzung erstellt wurde. Für WebSphere-Server auf Nicht-z/OS-Systemen ist dies nur in geclusterten Topologien von Bedeutung. Hier muss der Request zu dem Clustermember weitergeleitet werden, in dem auch die zugehörigen Objekte instanziert sind.

In z/OS-WebSphere-Instanzen spielt diese Information für die Gewichtung von Requests auch in ungeclusterten Installationen eine Rolle, die sich in obigen Antwortzeitverhalten widerspiegelt. In Kapitel 4 wurde bereits erklärt, wie Requests von der Control Region klassifiziert und dann an den Workload Manager übergeben werden. Diese Requests werden von WLM in einer Queue verwaltet, von der sie von den Servant Regions zur Bearbeitung ausgelesen werden. Es existiert zu jeder Serviceklasse eine globale WLM-Queue. Darüber hinaus besitzt noch jede Servant Region eine eigene, lokale Queue. Auf welcher Queue ein Request zur Bearbeitung abgelegt wird, hängt davon ab, ob dem Request eine `JSESSIONID`-Variable anhängt. Besitzt der Request eine `JSESSIONID`-Variable, wird er auf der WLM-Queue der über die Partition-ID adressierten Serverinstanz (Servant Region) abgelegt. Fehlt dem Request eine solche Variable, wird er auf der globalen WLM-Queue abgelegt.

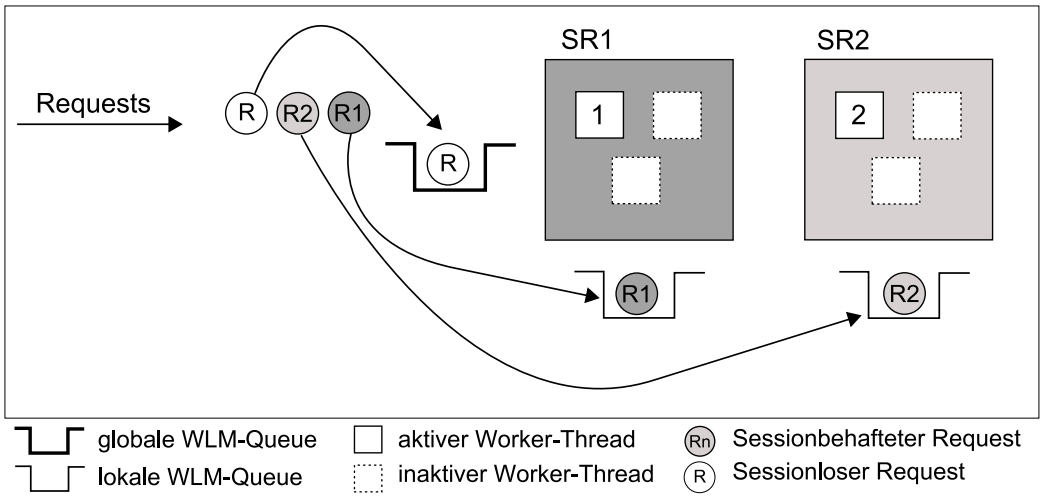


Abbildung 6.44: WLM-Verhalten bei sessionbehafteten und sessionlosen Requests.

Abbildung 6.44 veranschaulicht die unterschiedliche Behandlung von sessionlosen und sessionbehafteten Requests. R1 und R2 besitzen eine `JSESSION`-Variable und werden direkt auf der Queue der in der Variablen adressierten Servant Regions abgelegt. R ist ein noch sessionloser Request und wird auf der globalen, zur Serviceklasse gehörenden Queue abgelegt.

Man kann sich mit dem Programm *WLMQUE* ([IBM08c]) (*WLM Work Queue Viewer*) den Zustand der verschiedenen Queues anzeigen lassen. Um obige Aussage zu verifizieren wird dieses Programm in einem kleinen Testlauf verwendet.

Während des Testlaufs werden alle neun Requestsampler genau einmal durchlaufen. Vor und nach einem Testlauf wird der Zustand der Queues miteinander verglichen. Folgende Listings zeigen die Ausgabe von WLMQUE vor und nach dem Testlauf. Die Darstellung entspricht einem ISPF-Panel.

```
   Command ===>                                          Scroll ===> PAGE
                    Application Environment Monitor
Selection:  >HELP< >SAVE< >OVW< >ALL<
System: POC1      Sysplex: POC1      Version: z/OS 010800  Time: 12:19:36
ApplEnv_ Type SubName_ WMAS Del Dyn NQ QLen Str Hav Unb Trm Min_ Max__ ICnt
WACL1    CB   W1SR01   0087 No  Yes  1    0   0   1   0   0    6    12    0

WorkQue_ Del Wnt Hav ICnt QueIn_ QueOut QueLen QueTot__ Act_ Idl_
WASDFLT  No    1   1    0      0      0      0        1    0    6

SvAS Binding_ Ter Opr Btc Dem Have PEU_ ICnt WUQue____ Aff AffQue
0077 WASDFLT  No  No  No  No     6    6    0         1   0      0
```

Listing 6.3: Zustand der WLM-Queues vor einem Testlauf.

Neben dem Queue-Zustand zeigt das Programm noch weitere Informationen an, die für diese Untersuchung nicht von Bedeutung sind. Wichtige Werte sind hier der `WUQue__`-Wert und der `AffQue`-Wert. Der `WUQue__`-Wert zeigt die Summe der bis zu diesem Zeitpunkt auf der globalen Queue abgelegten Requests an. Der `AffQue`-Wert zeigt die Summe der auf der lokalen Queue abgelegten Requests.

Nachdem der Zustand wie oben gezeigt erfasst wurde, wird nun der Testlauf gestartet, in dem jeder Sampler genau einmal abgeschickt wird. Danach besitzen die Queues folgenden Zustand:

```
Command ===>                                          Scroll ===> PAGE
                      Application Environment Monitor
 Selection:  >HELP< >SAVE< >OVW< >ALL<
 System: POC1      Sysplex: POC1      Version: z/OS 010800  Time: 12:29:30
  ApplEnv_ Type SubName_ WMAS Del Dyn NQ QLen Str Hav Unb Trm Min_ Max__ ICnt
 WACL1    CB   W1SR01   0087 No  Yes  1    0   0   1   0   0    6    12    0

 WorkQue_ Del Wnt Hav ICnt QueIn_ QueOut QueLen QueTot__ Act_ Idl_
 WASDFLT  No    1   1    0      0      0      0        5    0    5

 SvAS Binding_ Ter Opr Btc Dem Have PEU_ ICnt WUQue____ Aff AffQue
 0077 WASDFLT  No  No  No  No     6    6    0         5   0      7
```

Listing 6.4: Zustand der WLM-Queues nach einem Testlauf.

Wie zu erwarten ist die Anzahl der Requests, die auf die lokale Queue gelegt wurden, auf sieben angestiegen. Wider Erwarten steigt die Anzahl der Requests auf der globalen Queue nicht auf drei sondern auf fünf. Woher die beiden überzähligen Requests kommen, konnte nicht nachhaltig geklärt werden.

Allerdings belegt dieser kleine Testlauf die oben geäußerte Vermutung, dass sessionbehaftete Requests auf den lokalen Queues der Servant Regions abgelegt werden.

Über eine interne WebSphere-Umgebungsvariable lässt sich nun kontrollieren, von welcher Queue eine Servant Region den als nächstes zu verarbeitenden Request abgreift. Diese Variable wird hier als `SELECT_POLICY` bezeichnet.

Je nach Wert der Variablen gilt eine der folgenden Regeln:

Variablenwert	**Verhalten**
0	Der älteste Request wird zuerst behandelt, unabhängig von welcher Queue.
1	Der Request auf der lokalen Queue, falls vorhanden, wird zuerst behandelt.
2	Der Request auf der globalen Queue wird immer zuerst behandelt.

Mit diesem Wissen und der Information darüber, wo sessionlose und sessionbehaftete Requests abgelegt werden, lässt sich nun herleiten, wie es zu obigem Antwortzeitverhalten kommt. Diagramm 6.45 zeigt das Antwortzeitverhalten im integrierten Szenario mit einem Prozessor und 300 gleichzeitigen Benutzern für alle drei möglichen Belegungen des Wertes `SELECT_POLICY`. Auf der y-Achse ist die durchschnittliche Antwortzeit in ms dargestellt. Die x-Achse zeigt die drei möglichen Belegungen der `SELECT_POLICY`-Variablen.

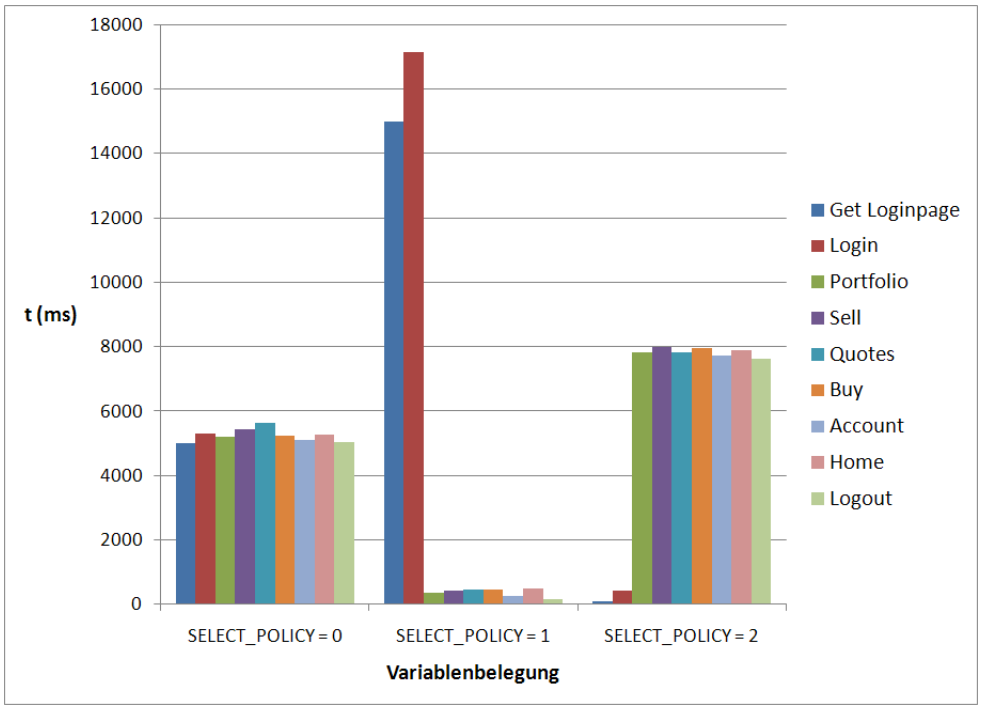


Abbildung 6.45: Auswirkung der Variable `SELECT_POLICY` auf das Antwortzeitverhalten.

Belegt man die Variable mit dem Wert 0, so gleicht das Antwortzeitverhalten des integrierten Szenarios dem des verteilten Szenarios. Die Verschlechterung findet gleichverteilt über alle Requests statt, da immer der älteste Request zuerst behandelt wird.

Die Belegung mit dem Wert 1 entspricht der Standardeinstellung in WebSphere. Requests auf der globalen Queue bleiben liegen, solange sitzungsaffine Requests auf den lokalen Queues vorhanden sind. Dies ist ein günstiges Verhalten, da Benutzer, die bereits eingeloggt sind und mit der Anwendung arbeiten, von einer Überlastung des Servers nicht betroffen sind. Sie können ihre Arbeit ungestört bei gutem Antwortverhalten des Servers fortführen.

Eine Belegung mit dem Wert 2 zeigt eine Umkehr des Verhaltens bei Wert 1. Sessionlose Requests, die auf der globalen Queue liegen, werden bevorzugt behandelt.

Diese Möglichkeit der Priorisierung von Requests ist in dieser Form nur bei Verwendung von z/OS als Betriebssystem verfügbar. Man kann hier die einzelnen Requests mit dem WLM-Subsystem über Serviceklassen unterschiedlich priorisieren (vgl. Kapitel 4.1.1, S.25 ff.). Über diese Priorisierung hinaus, lässt sich, wie eben gezeigt, über den Sessionzustand eines Requests eine weitere Priorisierung vornehmen. Diese Mechanismen führen zu einer erhöhten Stabilität der Anwendung im Überlastungsfall und zu einer garantierten Qualität des Services. Eine WebSphere-Installation auf Nicht-z/OS-Systemen lässt nur die Verwaltung von Adressräumen über Prioritätensteuerung des Betriebssystems zu. Das ist im Vergleich zum z/OS-Fall sehr grobgranular und ein entscheidender Nachteil.

6.4.3 Zusammenfassung der Auswertung der jMeter-Daten

Die gezeigten Daten über den Durchsatz und die durchschnittlichen Antwortzeiten der Requestsampler spiegeln die im Unterkapitel zuvor gemachten Erfahrungen über die Prozessorauslastung wider. Die größere Belastbarkeit des integrierten Szenarios auf z/OS äußert sich in einer höheren Durchsatzrate und einem schnelleren Antwortverhalten.

Darüber hinaus bietet z/OS als Betriebssystem mit dem Workload Manager einen Mehrwert, der im verteilten Szenario nicht gegeben ist. Es ist auf z/OS möglich bestimmten Nutzergruppen ein gleichbleibend gutes Antwortverhalten zu garantieren, selbst wenn der Server unter starker Überlastung leidet. Was für den Trade6-Benchmark im Zusammenhang mit HTTP-Sessions gilt, ist auch für jede andere Anwendung gültig, die HTTP-Sessions erstellt. Das ist ein starkes Argument für den Betrieb eines WebSphere Application Servers auf z/OS.

6.4.4 Multiservanten unter z/OS

Ebenfalls nur unter z/OS hat man die Möglichkeit innerhalb einer Serverinstanz mehrere JVMs, sprich Servant Regions, zu betreiben. Wie in Abschnitt 4.1, S. 25 ff., teilweise bereits erläutert, hat das mehrere Vorteile:

- **Isolation:** Unterschiedliche Anwendungen können innerhalb eines Servers in verschiedenen Adressräumen (JVMs) betrieben werden.
- **Ausfallsicherheit:** Der Ausfall einer Servant Region beeinträchtigt andere Servant Regions nicht zwangsläufig und der Server bleibt verfügbar.
- **Verschiedene Serviceklassen:** Die Servant Regions können bedarfsgerecht unterschiedlichen Serviceklassen zugewiesen werden. Das ist für die Nutzung des WLM-Subsystems und mehreren Serviceklassen zwingend notwendig (vgl. Kapitel 4, S. 27).

Diese Vorteile bezahlt man mit einem leicht gestiegenen CPU-Bedarf, wie Diagramm 6.46 zeigt. In einem Testlauf mit 100 gleichzeitigen Benutzern wird in drei Szenarien der CPU-Verbrauch erfasst.

- **z/OSx2 - 1 Servant Region:** Integriertes Szenario mit zwei dedizierten Prozessoren und einer Servant Region für Trade6.
- **z/OSx2 - 2 Servant Regions:** Integriertes Szenario mit zwei dedizierten Prozessoren und zwei Servant Regions für Trade6.
- **zLinux+z/OS:** Verteiltes Szenario mit WebSphere unter zLinux.

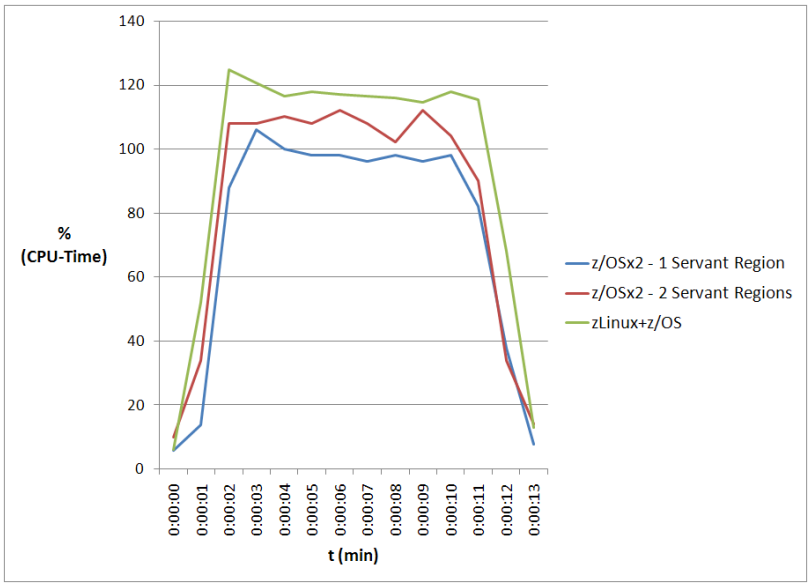


Abbildung 6.46: CPU-Verbrauch im verteilten und integrierten Szenario mit einem bzw. zwei Servanten.

Durch die Verwaltung von zwei Servant Regions und die in der Folge notwendigen Kontextwechsel zwischen den Adressräumen steigt die CPU-Auslastung des integrierten Szenarios leicht an. Sie bleibt aber unter der CPU-Auslastung des verteilten Szenarios und die Mehrbelastung ist vertretbar, wenn man die möglichen Vorteile in Betracht zieht.

Die Antwortzeiten steigen vernachlässigbar an und der Durchsatz der Anwendung verringert sich leicht, wie die folgenden beiden Diagramme zeigen.

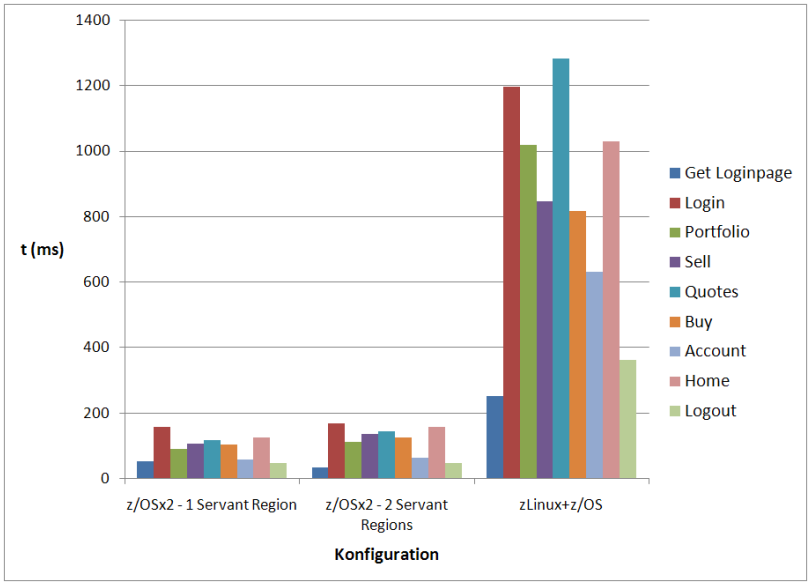


Abbildung 6.47: Antwortzeiten der Requestsampler bei Erhöhung der Servantenzahl.

Diagramm 6.47 zeigt die Antwortzeiten der verwendeten Requestsampler (in ms) für einen Testlauf mit 150 gleichzeitigen Benutzern. Die Erhöhung der Antwortzeit durch den zusätzlichen Servanten ist vernachlässigbar klein.

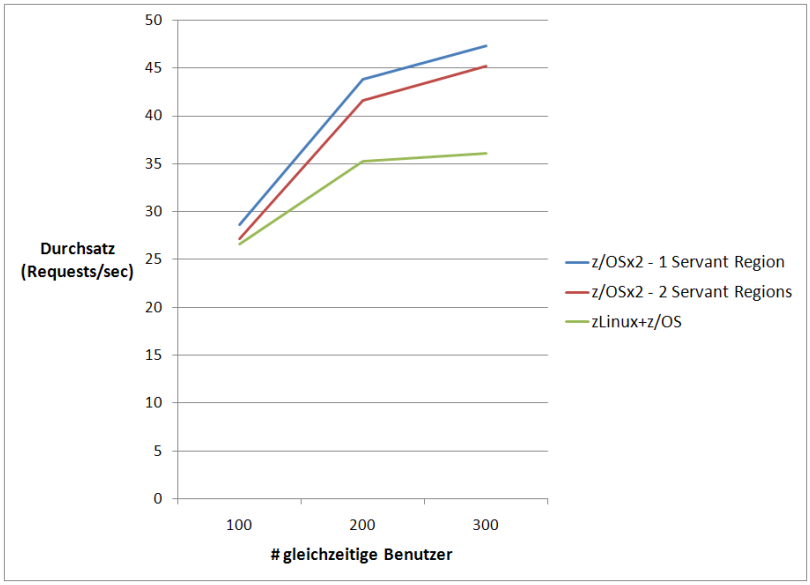


Abbildung 6.48: Veränderung der Durchsatzrate durch die Erhöhung der Servantenzahl.

Diagramm 6.48 zeigt die Durchsatzrate für das verteilte Szenario und das integrierte Szenario mit einem bzw. zwei Servant Regions. Durch den erhöhten Verwaltungsaufwand und die Kontextwechsel steigt die Prozessorbelastung an und der Durchsatz sinkt leicht. Er bleibt aber noch über dem Durchsatz des verteilten Szenarios.

6.5 Bewertung der Ergebnisse

Anhand der vorliegenden Ergebnisse kann eine Empfehlung ausgesprochen werden. Eine konsolidierte Lösung auf System z mit z/OS als Betriebssystem bietet aus technischer Sicht mehrere Vorteile.

Durch die Anbindung der Datenbank über einen JDBC-Treiber Typ 2 lässt sich der Verbrauch an CPU-Zeit um bis zu 20% reduzieren. Die so frei bleibende Prozessorleistung erhöht die Leistungsfähigkeit des Systems oder kann für andere Dienste genutzt werden.

Bei einer verteilten Konfiguration und der daraus resultierenden Datenbankverbindung über einen JDBC-Treiber Typ 4 verteilt sich die Gesamtlast ungleichmäßig auf die beteiligten Servermaschinen. Die gemachten Untersuchungen zeigen, dass das verteilte Szenario nicht in der Lage ist, beide Systeme voll auszulasten. Man könnte hier einen Ausgleich schaffen, indem man die Prozessorressourcen der zLinux-LPAR erhöht. Das setzt genaue Vorbetrachtungen über die zu erwartende Belastung der Server voraus. Diese ist in der Realität möglicherweise nur schwer genau vorherzusagen.

Die unterschiedliche Belastbarkeit zeigt sich auch in den Durchsatzwerten und dem Antwort-

zeitverhalten. Die konsolidierte Lösung ist durch die bessere Datenbankanbindung auch hier der verteilten Lösung überlegen. Die Softwarearchitektur des *WebSphere Application Servers for z/OS* bietet durch die Nutzung des z/OS-Subsystems *Workload Manager* weitere Vorteile für die Verwaltung der erstellten Clientsitzungen. Man hat nur hier die Möglichkeit einer feingranularen Priorisierung der Benutzer und deren Requests über Serviceklassen und die Konfiguration der WLM-Queues für die Serviceklassen und die Servant Regions.

6.5.1 Bewertung der Kosten

Bisher wurden nur die technischen Aspekte des Serverbetriebs mit unterschiedlichen Konfigurationen betrachtet. Bewertet man die vorliegende Untersuchung unter betriebswirtschaftlichen Aspekten, ändert sich das Bild ein wenig.

Analysten bewerten die Einstandskosten (*Total Cost of Aquisition - TCA*) für System z höher, als die TCA für verteilte Systeme auf Nicht-z-Basis. Sie sind sich aber fast unisono einig, dass System z eine bessere *Total Cost of Ownership* (*TCO*) für größere Installationen bietet, wenn man zum Beispiel die Kosten pro Transaktion betrachtet. Diese Kosten werden über so genannte *Specialty Engines* für bestimmte Anwendungen weiter optimiert.

IBM bietet zum derzeitigen Zeitpunkt fünf verschiedene Specialty Engines an, wovon die folgenden drei für diese Arbeit von Bedeutung sind ([EOO06]):

- **zIIP:** Der *System z9 Integrated Information Processor* ist für die Verarbeitung von Datenbanklast, die durch entfernte Anfragen im DDF-Adressraum und die daraus resultierenden Enclaven entstehen, konzipiert.
- **IFL:** Die *Integrated Facility for Linux* ist ein System z-Prozessor, bei dem ein oder zwei Instruktionen deaktiviert werden, die nur unter z/OS Verwendung finden. Er ist für die Ausführung des Linux-Betriebssystems gedacht.
- **zAAP:** Der *System z9 Application Assist Processor* ist für die Ausführung von Java-Programmen gedacht und hat einige Funktionen, wie z.B. Interrupt Handling , deaktiviert. Es kann kein Betriebssystem auf diesem Prozessor ausgeführt werden.

Im Prinzip handelt es sich bei allen Dreien um normale System z-Prozessoren, die hard- und/oder softwaregesteuert nur die Ausführung bestimmter Workloads zulassen.

Die Anzahl der Prozessoren in einem System z ist ein Teil der Berechnungsgrundlage für dessen Anschaffungs- und Betriebskosten. Die Specialty Engines fließen in die Summe der Prozessoren nicht mit ein. IBM macht auf diese Weise die System z für den Betrieb von Java-Workload, Datenbanken und Linux attraktiv.

Die Specialty Engines können für beide, hier untersuchten, Konfigurationen interessant sein. Für den Java-Workload im konsolidierten Betrieb kann ein zAAP verwendet werden. Im verteilten Betrieb ist die Datenbanklast auf einem zIIP ausführbar.

In einer Untersuchung der PPI Financial Systems GmbH wurde die Effizienz der Nutzung eines zAAPs für eine Home-Banking-Anwendung der Firma untersucht. Bei der Analyse ([Sys06]) stellte sich heraus, dass die untersuchte Anwendung mit einem Anteil von bis zu 85% auf einem zAAP ausgeführt wird. Die WebSphere Application Server-Infrastruktur selbst, bestehend aus den Control- und Servant-Regions, möglichen Deployment-Manager-Regions, Node Agents, usw. lässt sich sogar bis zu 90% auf einem zAAP ausführen.

Für den in dieser Arbeit verwendeten Trade6-Benchmark kann ca. 50% der Last auf einem zAAP ausgeführt werden, wie folgendes Diagramm zeigt. Die zur Erfassung verwendete Metrik ist *% AAP on CP*, welche auch ohne einen zAAP im Serverrechner anzeigt, wieviel Last auf einem zAAP ausgeführt werden würde.

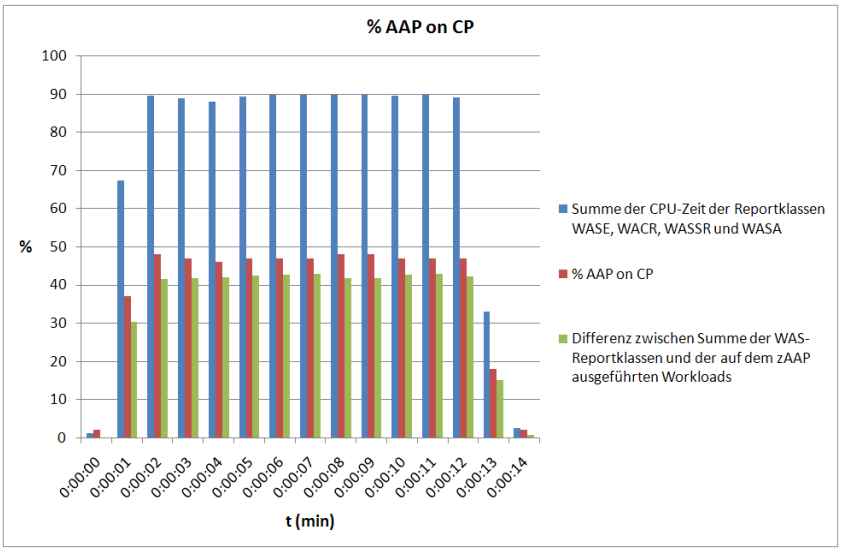


Abbildung 6.49: zAAP-Anteil an der Gesamtlast bei 150 gleichzeitigen Benutzern im integrierten Szenario mit einem Prozessor

Der blau dargestellte Zeitanteil entspricht der Last, die durch den WebSphere-Server verursacht wird. Der rote Anteil würde auf einem zAAP ausgeführt, wenn dieser im Serverrechner installiert wäre. Der grüne Anteil würde auch bei einem installierten zAAP auf einem *General Purpose Processor* ausgeführt werden. Dieser letzte Anteil ist im Vergleich zur Untersuchung der PPI Financial Systems GmbH relativ hoch, da in den WASE-Enclaven noch der Datenbankanteil der Transaktionen enthalten ist. Geht man davon aus, dass ca. 40-50% der CPU-Zeit der WASE-Enclaven in der Datenbank verbraucht wird, erhält man wieder die 80-90% Java-Workload, die sich auf einem zAAP ausführen lassen.

Mittels Specialty Engines lässt sich der betriebswirtschaftliche Nachteil eines System z gegenüber günstigeren Serverlösungen ausgleichen.

7 Zusammenfassung und Ausblick

Nach einer kurzen Zusammenfassung der vorliegenden Arbeit, wird anschließend ein Ausblick auf weitere, mögliche Untersuchungen gegeben.

7.1 Zusammenfassung

Zentrale Fragestellung dieser Arbeit war, wie sich die unterschiedliche Anbindung einer DB2-Datenbank an einen *WebSphere Application Server* im Verbrauch an CPU-Zeit widerspiegelt. Unterschiede diesbezüglich wurden im Laufe einer Anwendungsanalyse durch die Firma IBM beobachtet. In dieser Arbeit sollten diese Unterschiede verifiziert und quantifiziert werden.

Zu diesem Zweck wurde in einem ersten Schritt eine Versuchsumgebung auf einem zSeries-Rechner an der Uni Leipzig installiert (vgl. Kapitel 5, S.45 ff.). Diese Testumgebung umfasst zwei Systemkonfigurationen:

- **Integrierte Konfiguration:** In der integrierten Konfiguration befinden sich der *WebSphere Application Server V6.1 for z/OS* und die *IBM DB2 for z/OS V8*-Datenbank in derselben z/OS-LPAR. Die Anbindung der Datenbank geschieht über einen JDBC-Treiber Typ 2. Das bedeutet, die Kommunikation zwischen Anwendungsserver und Datenbank findet über native Datenbankbibliotheken innerhalb des Hauptspeichers der LPAR statt.

- **Verteilte Konfiguration:** Die DB2-Datenbank verbleibt auch in dieser Konfiguration in der z/OS-LPAR. Der *IBM WebSphere Application Server Network Deployment V6.1* befindet sich innerhalb einer zweiten LPAR. In dieser LPAR ist ein *SuSE Enterprise Linux Server V9R3* als Betriebssystem installiert. Der Zugriff auf die Datenbank findet über einen JDBC-Treiber Typ 4 statt. Anfragen an die Datenbank werden über ein TCP/IP-Netzwerk gestellt, was eine Serialisierung der Anfrage- und Rückgabeparameter notwendig macht.

Auf Basis dieser Servertopologien wurde in beiden WebSphere-Servern die *Trade6*-Benchmarkanwendung installiert (vgl. S.51ff.). Der Trade6-Benchmark ist speziell für Performanceuntersuchungen konzipiert und simuliert eine fiktive Aktienhandelsplattform. Er bietet drei Zugriffsmechanismen: HTTP-Clients, Java-Clients und WebServices.

Die fiktiven Benutzer- und Aktiendaten des Benchmarks wurden in der DB2-Datenbank angelegt. Der Zugriff auf die Datenbank findet über die jeweils konfigurierten JDBC-Treiber des Typs 2 bzw. 4 statt.

Die auf der Testanwendung durchgeführten End-To-End-Tests sollten einer realen Benutzung entsprechen. Um diese zu simulieren, wurden im Lasterzeugungstool *jMeter* Nutzungsszenarien konfiguriert (vgl. S.56 ff.). jMeter ist dabei auf einem Clientrechner installiert und löst über HTTP-Request in der Benchmarkanwendung fiktive Aktionen der Nutzer aus. In beiden Systemkonfigurationen wurden anschließend mehrere Testläufe mit steigender Anzahl der gleichzeitig auf die Anwendung zugreifenden Benutzer durchgeführt. Durch die steigende Anzahl der gleichzeitigen

Benutzer wurde eine steigende Belastung der Server bis hin zur Saturierung der Konfigurationen simuliert.

Zur Überwachung der Server wurde *RMF Performance Monitoring*, kurz *RMF-PM*, der Firma IBM eingesetzt (vgl. S.58 ff.). Dieser verwendet eine Client-Server-Architektur zur Erfassung von Leistungsdaten der Servermaschinen. RMF-PM erhält über TCP/IP von einem Data-Gatherer zuvor im Server aufgezeichnete und akkumulierte Daten. RMF-PM wurde eingesetzt, da es für beide verwendeten Betriebssysteme eine Data-Gatherer-Komponente gibt. Für z/OS ist dies der *Monitor III* des *Resource Measurement Facility*-Subsystems. Für Linux stellt IBM dem *RMF Linux Data Gatherer* zur Verfügung. Der Data-Gatherer für Linux lieferte allerdings im Verlauf der Tests teilweise fehlerhafte Daten (vgl. S.88 f.). Er wird derzeit von IBM auch nicht mehr weiterentwickelt. Für den Großteil der vorgenommenen Messungen waren die gelieferten Daten allerdings korrekt und konnten für eine Auswertung herangezogen werden.

Die erfassten Leistungsdaten bestätigten die von IBM gemachten Beobachtungen. Eine konsolidierte Anbindung der Datenbank über einen JDBC-Treiber Typ 2 benötigt ca. 20% weniger CPU-Zeit als eine verteilte Architektur mit Anbindung der Datenbank über einen JDBC-Treiber Typ 4 (vgl. S.97 ff.).

Darüber hinaus zeigte sich, dass sich bei einer verteilten Konfiguration die Gesamtlast nicht gleichmäßig auf beide Servermaschinen verteilt. Die Anfragen des WebSphere-Servers in der ausgelasteten Linux-LPAR konnten den Prozessor der Datenbank-LPAR nicht voll auslasten. Daraus resultierte eine größere Leistungsfähigkeit der integrierten Konfiguration, welche die gegebenen Prozessorressourcen vollständig nutzen konnte.

Eine WebSphere-Instanz auf z/OS besteht, aufgrund der Verwendung des z/OS-eigenen *Workload Manager*-Subsystems, aus mehreren Prozessen (vgl. S. 25 ff.). Die Klassifizierung von Requests durch die *Control Region* und die anschließende Weiterleitung an die entsprechende WLM-Queue (vgl. S. 104 f.) führt dazu, dass eine WebSphere-Instanz auf z/OS eine qualitative Unterscheidung von Requests vornehmen kann.

Benutzer können anhand ihrer äußeren Eigenschaften, wie IP, Benutzerkennung, Standort, etc, einer Serviceklasse zugewiesen werden. Durch die Zuordnung von Serviceklassen kann eine feingranulare Priorisierung der Benutzer vorgenommen werden, die in dieser Form auf anderen Plattformen nicht möglich ist.

Weiterhin garantiert diese Unterscheidung bestimmten Benutzern der Anwendung selbst bei Überlastung des Servers ein gutes Antwortzeitverhalten. Zum einen lassen sich wichtigere Benutzer in höher priorisierte Serviceklassen einordnen, so dass die mit ihnen verbundenen Transaktionen bevorzugt Ressourcenzugriff innerhalb der Servermaschine erhalten. Zum anderen können Benutzer mit bereits bestehenden HTTP-Sessions bevorzugt vom Server bedient werden (vgl. S. 103 ff.). Dieser Mechanismus ließe sich nutzen, um neu hinzukommende Benutzer auf die Überlastung des Servers hinzuweisen. Für Installationen des WebSphere-Servers auf anderen Systemen ist dies nicht möglich. Da hier nur auf Prozessebene priorisiert werden kann, wirkt sich eine Überlastung des Systems gleichverteilt auf alle Nutzer aus.

7.2 Ausblick

Die qualitative Unterscheidung der Requests anhand ihres Sessionzustands zeigt sich bisher nur im Antwortzeitverhalten des Servers. Diese Eigenschaft ließe sich aber auch programmatisch nutzen,

um den Überlastungszustand des Servers an anfragende Clients zu übermitteln.

Weiterhin fanden die gemachten Untersuchungen weitestgehend ohne vorherige Optimierungen statt. Es gibt eine große Anzahl an Konfigurationsmöglichkeiten, die Einfluss auf die Prozessorauslastung und die Performanz des WebSphere-Servers haben können.

- **Caches:** Es werden verschiedene Caches in einer WebSphere-Installation eingesetzt. So werden zum Beispiel die Ergebnisse von Datenbankanfragen in einem Cache auf dem Anwendungsservers zwischengespeichert. Das verteilte Szenario kann von einer Optimierung dieses Caches profitieren.

- **JVM:** Die Heapgröße der *Java Virtual Machine*, in denen die WebSphere-Server instanziert sind, oder das Verhalten des *Garbage Collectors* können optimiert werden.

- **Verbindungs-Pools:** Auf Datenbankseite lässt sich über *Connection Concentration* oder *Connection Pooling* die Kommunikation mit dem Anwendungsserver optimieren.

- **Worker-Threads:** Im WebSphere-Server, wie auch in der Datenbank, kann die Anzahl der Workerthreads variiert und an die installierte Anwendung angepasst werden.

- **Verwendung anderer Clients:** In der gemachten Untersuchung wurden Anfragen an den Server nur über HTTP-Requests gestellt. In weiteren Untersuchungen könnten weitere Zugriffsarten, wie zum Beispiel *WebServices*, und ihre Auswirkung auf die Performanz untersucht werden.

Dies sind nur einige der wichtigsten Punkte, die als Grundlage für weitere Untersuchungen in Betracht kommen. Die Architektur verteilter Anwendungen und ihre Optimierung bleiben auch in Zukunft ein breites und interessantes Forschungsgebiet in der Informatik.

A Inhalt der beigefügten CD

Der Inhalt der beigefügten CD ist wie folgt aufgebaut:

- **jmeterkonfiguration/**

  Hier befinden sich die jMeter-Konfigurationsdateien für die unternommenen Testläufe. In den jeweiligen Verzeichnissen zu den Testläufen finden sich auch die erfassten Daten im Tabellendatenformat.

- **requestmetrics/**

  In diesem Verzeichnis liegen die erfassten Request Metrics zu den in den Testläufen verwendeten Requestsamplern.

- **diplomarbeitthomasbitschnau.pdf**

  Die vorliegende Diplomarbeit im *Portable Document Format* (*PDF)*. Um diese Datei öffnen zu können wird *Adobe Reader* benötigt, welcher hier heruntergeladen werden kann:
  http://www.adobe.com/products/acrobat/readstep2.html

Literaturverzeichnis

[BED+04] Ann Black, Mike Everett, Dave Draeger, et al. *IBM WebSphere Application Server for Distributed Platforms and z/OS - An Administrator's Guide*. IBM Press, 2004.

[CG+06] Piere Cassier, Keith George, et al. *Redbook: z/OS Parallel Sysplex Configuration Overview*, 2006.

[CJC06] Mike Cox and et al. Jim Cunningham. *Redbook: IBM WebSphere Application Server for z/OS Version 6: A performance report*, 2006.

[CK+05] Pierre Cassier, Raimo Korhonen, et al. *Redbook: Effective zSeries Performance Monitoring using Resource Measurement Facility*, 2005.

[EH+07] Mike Ebbers, Christopher Hastings, et al. *Introduction to the New Mainframe: Networking*, 2007.

[EOO06] Mike Ebbers, Wayne O'Brian, and Bill Ogden. *Redbook: Introduction to the New Mainframe: z/OS Basics*, July 2006.

[Ham05] Ulrike Hammerschall. *Verteilte Systeme und Anwendungen*. Pearson Studium, 2005.

[IBM06] IBM. *Trade6 - Technical Documentation*, 2006. Die Dokumentation ist Teil der Anwendung selbst.

[IBM07] IBM. *IBM Developer Kit and Runtime Environment, Java Technology Edition, Version 5.0, Diagnostics Guide*, 2007.

[IBM08a] IBM Information Center - WebSphere Application Server V6.1, 15.01.2008. http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.base.doc/info/welcome_base.html.

[IBM08b] IBM Education Assistant - IBM WebSphere Application Server V6 - z/OS Architecture, 15.01.2008. http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.was_v6/was/6.0.1/BigPicture/WASv601_zOS_Architecture/player.html.

[IBM08c] IBM WLM for z/OS and OS/390 - WLM Work Queue Viewer, 15.01.2008. http://www.ibm.com/servers/eserver/zseries/zos/wlm/tools/wlmque.html.

[JDB08] Überblick über die JDBC-Architektur, 15.01.2008. http://java.sun.com/products/jdbc/overview.html.

[JEE08a] Webseite zur Java Platform, Enterprise Edition von SUN, 15.01.2008. http://java.sun.com/javaee/.

[JEE08b] Tutorial zur Java Platform, Enterprise Edition von SUN, 15.01.2008. http://java.sun.com/javaee/5/docs/tutorial/doc/.

[jMe08] Dokumentation des jMeter-Projektes, 15.01.2008. http://jakarta.apache.org/jmeter/usermanual/.

[JSE08] Webseite zur Java Platform, Standard Edition von SUN, 15.01.2008. http://java.sun.com/javase/.

[KC+06] Alex Louwe Kooijmans, Tony J. Cox, et al. *Redbook: Performance Monitoring and Best Practices for WebSphere on z/OS*, August 2006.

[Lan04] Thorsten Langner. *Web-basierte Anwendungsentwicklung*. Elsevier - Spektrum Akademischer Verlag, 2004.

[Lyo07] Hélène Lyon. Vortrag: Soa and z/os - the perfect match, 2007.

[Sad06] Carla Sadtler. *Redbook: WebSphere Application Server V6.1:Technical Overview*, 2006.

[SB+05] Bart Steegmans, Carsten Block, et al. *Redbook: DB2 for z/OS and WebSphere: The Perfect Couple*, 2005.

[Sch07] Thomas Schulze. *Vortrag: z Optimized Applications: Performance and Function Testing*, 2007.

[Sha03] Bill Shannon. *Java™ 2 Platform Enterprise Edition Specification, v1.4*, 2003.

[Spr06] Prof. Dr.-Ing. Wilhelm G. Spruth. *Skriptum zu Vorlesung „Client/Server Systeme“*, 2006.

[Sta05] Thomas Stark. *J2EE - Einstieg für Anspruchsvolle*. Addison-Wesley, 2005.

[Sys06] PPI Financial Systems. *Betriebskosten reduzieren durch Umstieg auf Java-basierte z/OS-Anwendungen*, 2006.

[uAD+07] Pierre Cassier und Annamaria Defendi et al. *Redbook: System Programmer's Guide to: Workload Manager*, 2007.

[uFA+06] Carla Sadtler und Fabio Albertoni et al. *Redbook: WebSphere Application Server V6.1: System Management and Configuration*, November 2006.

[vB07] Joachim von Buttlar. Workshop: System z Architecture, Mai 2007.

[Yu08] Linfeng Yu. Artikel: Understanding WebSphere for z/OS, 15.01.2008. http://websphere.sys-con.com/read/98083.htm.