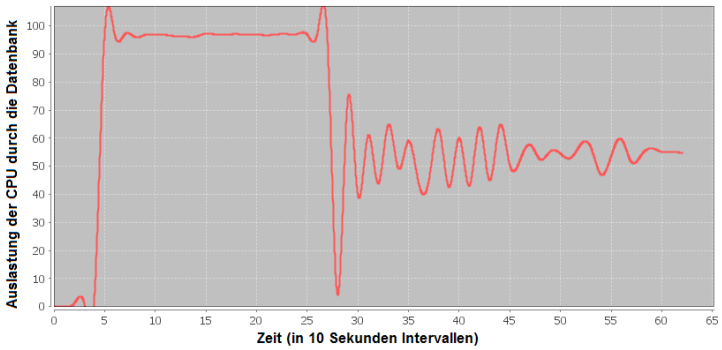
- B a c h e l o r a r b e i t -

Studiengang Medieninformatik (BA)

Hochschule der Medien Stuttgart

Entwicklung einer clientseitigen Schicht zur Lastkontrolle auf DB2 z/OS



Erstprüfer: Prof. Dr. Martin Goik

Zweitprüfer: Stephan Arenswald (Dipl.-Inform.)

WS 2011/2012

Thomas Uhrig (Matr. 20201)

Schwabstraße 86

70193 Stuttgart

- IBM Confidential -

Erklärungen

Marken der IBM

Diese Arbeit enthält Produktnamen (u.a. Omegamon, z/OS, DB2, Tivolie und Resource Measurement Facility), die eingetragene Marken der International Business Machines Corporation (IBM) in den USA und/oder anderen Ländern sind. Sie unterliegen somit gesetzlichen Bestimmungen. Diese können unter `http://www.ibm.com/legal/de/de/` eingesehen werden.

Vertraulichkeit der Arbeit

Diese Arbeit entstand in Zusammenarbeit mit der IBM Deutschland Research & Development GmbH in Böblingen. Daher unterliegen sämtliche Inhalte einer Sperrfrist und sind bis zum Ablauf dieser vertraulich. Diese Sperrfrist endet nach drei Jahren, also am 01. Januar 2015.

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Abschlussarbeit im Studiengang Medieninformatik (BA) der HdM Stuttgart, eigenständig und ausschließlich unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt zu haben. Sämtliche Stellen, die wörtlich oder sinngemäß zitiert sind, wurden deutlich gekennzeichnet und finden sich im abschließenden Literaturverzeichnis wieder.

Thomas Uhrig, Stuttgart im Winter 2011.

_____

Abstrakt

Gegenstand dieser Arbeit ist die Implementierung einer clientseitigen Softwareschicht zur Lastkontrolle einer DB2 Datenbank auf dem Großrechnerbetriebssystem z/OS.

Die in der Programmiersprache Java entwickelte Applikation ermöglicht es, die Auslastung von frei wählbaren Ressourcen zu beschränken. Ein Beispiel hierfür ist die Auslastung der CPU, die einen gegebenen Grenzwert nicht überschreiten soll. Die Lastkontrolle geschieht auf Grundlage von aktuellen Leistungsdaten des z/OS-Systems, sowie anhand eines Regelwerks. Zugriffe einer Client-Anwendung auf eine Datenbank werden entsprechend dieser Vorschriften sofort zugelassen oder je nach Auslastung verzögert.

Die Softwareschicht kann ohne Änderungen für die eigentliche Applikation transparent auf dem Client eingerichtet werden. So können bestehende Programme erweitert werden, ohne sie modifizieren zu müssen. Die konkreten Java-Klassen, auf die Einfluss genommen werden soll, spielen dabei keine Rolle - sie können im Regelwerk beliebig definiert werden.

Als Ansatzpunkt für die Entwicklung dient die JDBC-Schnittstelle. Diese wird durch Instrumentierung zur Laufzeit für die Lastkontrolle zugänglich gemacht. Für das Abfragen der z/OS-Leistungsdaten wird die IBM Resource Measurement Facility genutzt. Die Regeln können vom Nutzer frei in XML verfasst und während der Ausführungszeit der Anwendung angepasst werden. Die Lastkontrollschicht bietet zudem verschiedene Reporting-Mechanismen, um ihre Arbeit zu überwachen - beispielsweise durch CSV-Dateien oder generierte Statistiken.

Im Folgenden wird zunächst die Problemstellung sowie ein fiktives Anwendungsszenario vorgestellt. Danach wird ein kurzer theoretischer Überblick zu Ressourcen und Lastkontrollen im Allgemeinen gegeben. Im anschließenden Hauptteil der Arbeit wird die Entwicklung des Lastkontrollsystems in Java vorgestellt. Am Ende finden sich Testläufe und Messungen zur Validierung des entwickelten Systems. Außerdem werden Ausblicke für zukünftige Entwicklungen gegeben.

Schlagwörter: z/OS, Lastkontrolle, Java, JDBC, DB2

Abstract

This paper presents an implementation of a client-side software layer for load management of a DB2 database on the host operation system z/OS.

The application, written in Java, enables control of the utilization of arbitrary resources. An example is the CPU utilization, that should be below a given boundary value. The load management is based on current performance data of the z/OS system, as well as a system of rules. Client-application requests to a database will be immediately processed or delayed, according to the current utilization and these rules.

The software layer can be deployed transparently on a client, without any changes to an existing program. Therefore, existing programs can be extended without any modification needed. The affected Java-classes are defined in the rule-file.

The starting-point for the development is the JDBC-API, which is made accessible at run-time by using Java instrumentation. The IBM Resource Measurement Facility is used to access performance data in the z/OS system. The system of rules can be specified by the user in an XML-file and can be changed during the run-time of the application. Thus, rules can be easily tuned and adapted to the actual use-case. The load management layer also provides different reporting mechanisms to monitor the execution - for example CSV-files or generated statistics.

The following text introduces the problem as well as a virtual use-case. A short theoretical overview about resources and load management is then given. In the main part of the thesis, the development of the load management layer in Java is described. This is followed by tests and measurements for the validation of the system as well as an outlook for further prospects.

Keywords: z/OS, Load Management, Java, JDBC, DB2

Vorwort

Die vorliegende Abschlussarbeit entstand im Studiengang Medieninformatik (Bachelor) der Hochschule der Medien Stuttgart, in Zusammenarbeit mit der IBM Deutschland Research & Development GmbH. Ziel der Abschlussarbeit ist die Entwicklung einer clientseitigen Softwareschicht zur Lastkontrolle einer DB2 Datenbank auf z/OS.

Herzlicher Dank geht an dieser Stelle an *Peter Mailand* für seine zahlreichen Tipps rund um die RMF. Dank geht ebenfalls an *Karoline Kasemir* für ihre Ratschläge zur RMF und vor allem für den Crashkurs in z/OS. *Karin Steckler* und *Marco Misere* gilt mein Dank für die zuverlässige Hilfe, falls sich Join-Abfragen wieder einmal im Nirgendwo verloren hatten oder Datenbankpasswörter und Berechtigungen fehlten. Dem gesamten *Team des OMEGAMON Performance Experts* gilt gleichsam mein Dank für die (mittlerweile) über einjährige Unterstützung in Praxissemester, Werksstudententätigkeit und Bachelor-Arbeit.

Besonderer Dank geht an meinen Betreuer *Stephan Arenswald*. Er stand mir während meiner gesamten Zeit bei der IBM jederzeit mit viel Engagement zur Seite.

Ebenso möchte ich mich bei meinem betreuenden Professor, *Dr. Martin Goik,* von der HdM Stuttgart bedanken.

Mein abschließender Dank geht an meine Familie und Freunde, die mich während des gesamten Studiums unterstützt haben. Euer Rückhalt hat mir sehr geholfen.

Thomas Uhrig, Stuttgart im Winter 2011.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

Teil I.

Problemstellung

1. Problemstellung

1.1. Ein Szenario aus der Praxis

Last kann ein System zum Erliegen bringen. Diese Situation hat sicherlich schon jeder am eigenen Rechner miterlebt. Prozesse laufen im Hintergrund, Programme sind gestartet und das Aufrufen des Browser bringt das hoffnungslos in Not geratenen Gerät völlig zum Stillstand. Mausklicks oder Tastaturbefehle werden (wenn überhaupt) erst nach Sekunden beantwortet und man gewinnt den Eindruck, dass der PC mehr mit sich selbst beschäftigt ist, als für den Nutzer zu arbeiten.

Solche Szenarien treten nicht nur im Heimanwenderbereich auf, sondern auch im professionellen Umfeld - spätestens dann, wenn ca. vierzig bis fünfzig Softwareentwickler an drei Standorten sich gerade zehn Testmaschinen teilen. Eventuell genügt hier schon ein einziger gestarteter Workload, um eine Maschine an ihre Leistungsgrenzen zu bringen. Dies blockiert die Maschine für Zugriffe, und zwar meist nicht nur für andere Teammitglieder, sondern auch für den jeweiligen Entwickler selbst. Denn betroffen von der Auslastung ist beispielsweise auch die Performance seiner Entwicklungstools, die auf diesem System laufen.

Dieses Beispiel ist keinesfalls fiktiv, sondern tritt häufig auf: Der IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS[1], ein Produkt zur Überwachung von DB2 Leistungsdaten auf z/OS, wird von einem knapp fünfzigköpfigen Team an mehreren Standorten weltweit entwickelt. Überwiegend findet die Arbeit zwar in Deutschland am Standort Böblingen statt, jedoch befinden sich Teile des Teams auch im weißrussischen Minsk und an mehreren Lokationen in den USA. Hierzu gehören neben Programmierern, Managern und Designern natürlich auch Tester, welche die Maschinen und vor allem die darauf laufenden Datenbanken besonders beanspruchen.

[1] im Weiteren kurz *Performance Expert*

Probleme wie dieses werden auf verschiedene Arten gelöst werden. Denkbar sind etwa spezielle Konfigurationen im z/OS-System selbst oder Workloads, die so konzipiert sind, dass sie das System nicht überlasten. Dies führt aber, wie im Falle des Performance Expert Teams, am Entwicklungsalltag vorbei. Denn oftmals haben auch Teammitglieder keine Administrationsrechte für die Zielsysteme[2] und Workloads sind häufig schon in großer Zahl vorhanden. Zudem sind flexible Lösungen gefragt, da statische Konfigurationen an Client und Server zu starr für den Softwareentwicklungsprozess sind.

Wie also für eine dynamische Lastkontrolle sorgen?

1.2. Kernproblem

Beim Operieren auf einer Datenbank wird durch eine Applikation *Last*[3] auf dieser erzeugt. Anfragen des Clients an das Datenbanksystem müssen entgegengenommen, bearbeitet und beantwortet werden. Hierzu ist es beispielsweise notwendig, Zugriffspläne zu erstellen, Datensätze zu puffern oder Aggregationen zu berechnen. All dies führt zum Verbrauch von Ressourcen - also zu Last auf das System. Da die beanspruchten Ressourcen jedoch limitiert sind, kann - und wird - dies zu Problemen führen: Die Antwortzeiten von Systemen können deutlich steigen (vgl. hierzu beispielsweise Millsap, 2010) oder das betroffene System ist überhaupt nicht mehr zu erreichen.

Die Applikationen selbst nehmen auf diesen Sachverhalt meist keine Rücksicht. Lastkontrolle obliegt für gewöhnlich dem Serversystem selbst, was in den meisten Fällen auch Sinn ergibt (eine Diskussion hierzu findet sich im Abschnitt 4.4). Gerade für Testzwecke ist eine umfangreiche Konfiguration am Zielsystem aber meist zu aufwändig und nur bedingt sinnvoll - Testfälle sind kurzlebig und können sich häufig ändern, Teammitgliedern fehlt die Berechtigung oder es kommt zu Konflikten um die konkreten Einstellungen. Zudem verlangen gerade umfangreiche Tests vielseitige Workloads[4] als Grundlage. Somit ist es für das Beispiel der Workloads schwer, *eine* generelle Konfiguration zu finden und diese bei Bedarf schnell zu ändern. Blockieren diese Workloads jedoch das System, führt das für die Entwickler wiederum zu Problemen (etwa bei der Leistung serverseitiger Tools).

Daher ist eine einfache Möglichkeit, die Lastentwicklung von (Test-) Workloads für bestimmte Leistungsdaten nach oben hin zu beschränken, wünschenswert.

[2] Dies ist in der Praxis üblich, z.B. um Kundensituationen realistischer nachzuempfinden.
[3] siehe Kapitel 3
[4] Hierbei ist „vielseitig" im Sinne von heterogenen Workloads, die eine möglichst hohe Bandbreite von Szenarien abdecken, zu verstehen.

1.3. Ziele

Ziel dieser Arbeit ist, eine Lösung für den zuvor beschriebenen Problemfall zu entwickeln. Dies soll anhand eines clientseitigen Lastkontrollsystems in Java umgesetzt und validiert werden. Es soll exemplarisch für die DB2 Datenbank auf z/OS implementiert werden, da diese für das Team des Performance Experts von (alleiniger) Bedeutung ist. Es wird angestrebt, eine Kontrollschicht zu erstellen, welche transparent für bisherige Anwendungsprogramme auf dem Client eingerichtet werden kann. Ihr Konfigurationsaufwand soll minimal sein und auch von Teammitgliedern ohne Administrationsrechte auf dem Zielsystem lokal vorgenommen werden können. Die Anforderungen an das System sind daher:

- ▷ Eine **transparente Installation** auf dem Client, ohne Änderungen an bestehenden Anwendungen.
- ▷ Eine Lastkontrolle anhand von **aktuellen Leistungsdaten** des z/OS Systems.
- ▷ Möglichkeiten zur individuellen **Konfiguration** (Regeln, Reporting, etc.).

Die Lastkontrollschicht soll dabei modular aufgebaut sein. Abbildung 1.1 zeigt anhand eines UML-Component-Diagramms, wie sich die Lastkontrolle in die vorhandene Infrastruktur einfügen soll.

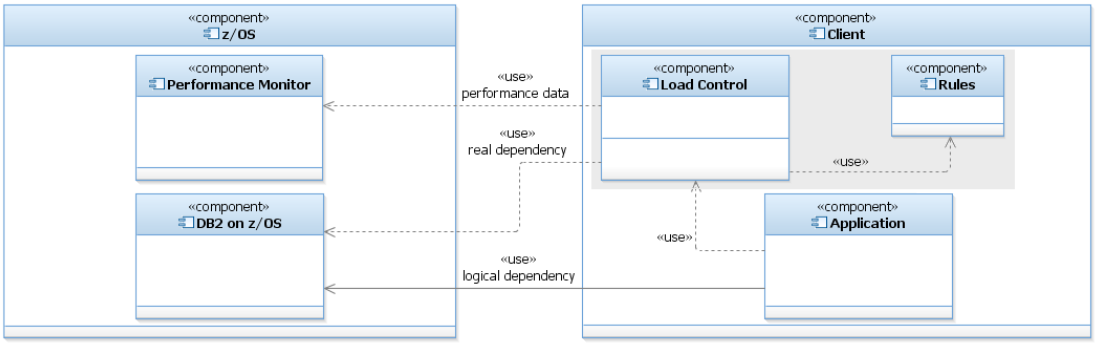


Abbildung 1.1.: UML-Diagramm der Infrastruktur mit der Lastkontrollschicht

Der Ansatz soll die von Workloads erzeugte Last auf einfache Weise lokal regulieren. So sollen „Freiräume" für andere serverseitige Tools garantiert werden. Die Funktion der Applikation wird abschließend in Abschnitt 11 evaluiert.

2. Abgrenzung von verwandten Themen

2.1. Serverseitige und zwischengeschaltete Lastkontrollen

Lastkontrollen können an drei Stellen eines Systems auftreten. Am häufigsten sind sie auf dem Server selbst zu finden. Der Begriff *Server* ist dabei im weitesten Sinne zu verstehen, da sowohl ein Betriebssystem als auch eine darauf laufende Applikation, jeweils für sich Last managen kann. Somit können beide Komponenten als Server verstanden werden. Serverseitige Lastkontrolle bedeutet also, dass das System, auf dem die Last auftritt, sich selbst darum kümmert, diese zu verwalten.

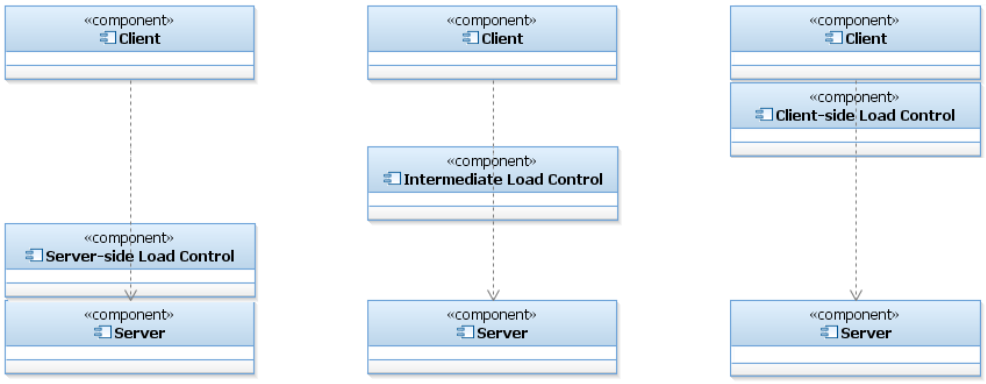


Abbildung 2.1.: Arten der Lastkontrollen

Ebenfalls denkbar sind zwischengeschaltete Lastkontrollen. So findet man z.B. häufig in Webserverarchitekturen eine (transparente) Firewall zwischen Client und Server, die die Zahl der Anfragen beschränkt[1]. Solche Firewalls dienen etwa dem Schutz vor DoS-Attacken[2], indem sie die Last auf den Server kontrollieren. Weder aber der Client, noch

[1] Eine Firewall dient zwar vorrangig anderen Aufgaben als der Lastkontrolle (z.B. dem Abweisen von speziellen IP-Adressen), sie kann aber auch für diese Zwecke genutzt werden. Da eine Firewall den zentralen Zugang zu einem System darstellt, können hier verschiedenste Dienste angesiedelt sein: Reporting, Session-Handling, Load Balancing oder Lastkontrollen. Man spricht hierbei von einem Reverse Proxy, eine[m] hochsicheren Rechner [...], der als erster Knoten der Infrastruktur Requests [...] in Empfang nimmt (Kriha and Schmitz, 2008, Seite 231).

[2] Ein *Denial of Service* Angriff versucht einen Webservern durch eine hohe Zahl von Anfragen zu überlasten und so zum Erliegen zu bringen.

der Server kennt diese Barriere. Auch beruhen solche Kontrollen häufig auf statischen Regeln und beziehen echte Leistungsdaten kaum mit ein. Vielmehr setzten sie Limits, beispielsweise wie oft eine bestimmte IP-Adresse pro Zeiteinheit einen Request an den Server schicken darf.

In dieser Arbeit soll jedoch eine dritte Art von Lastkontrolle betrachtet werden, nämlich auf der *Clientseite*. Sie unterscheidet sich von der serverseitigen dahingehend, dass sie nur auf sich selbst Einfluss nehmen kann. Etwaige andere Teilnehmer oder den Server selbst kann sie nicht beeinflussen. Dieser Umstand wird im Abschnitt 4.4 diskutiert. Von der zwischengeschalteten Lastkontrolle unterscheidet sie sich ebenfalls durch den Umstand, dass sie andere Teilnehmer und deren Requests nicht kontrollieren kann. Dafür kennt die clientseitige Lastkontrolle die eigene Applikation und deren (zukünftige) Anfragen besonders gut. Sie kann daher (im Gegensatz zur serverseitigen und zwischengeschalteten Lastkontrolle) wesentlich feingranularer auf die eigene Anwendung einwirken.

2.2. Load Balancing

Load Balancing bezeichnet das Verteilen von Last auf mehrere Komponenten. Diese Technik wird angewendet, um die Performance eines Systems zu erhöhen. Dabei besteht ein System nicht aus einer homogenen Komponente, sondern aus mehreren eigenständigen. Eine zentrale Instanz verteilt eintreffenden Requests gleichmäßig auf diese Komponenten.

Eine Webapplikation kann beispielsweise nicht nur auf einem einzigen, sondern auf mehreren Servern gehostet sein. Je nach Last werden Anfragen auf diese verteilt. Dies erhöht die Leistung des Systems sowie dessen Ausfallsicherheit und Verfügbarkeit. Dies kann nach statischen Regeln (z.B. dem Round-Robin-Verfahren) oder dynamisch, je nach aktueller Auslastung der Komponenten geschehen (Kriha and Schmitz, 2008, Seite 72).

Die hier vorgestellte Technik hingegen verteilt keine Lasten. Sie passt vielmehr die Arbeitsgeschwindigkeit einer lokalen Anwendung an die Leistung eines Servers an. Die Last wird also nicht verteilt, sondern verzögert und über die Zeit „portioniert“.

2.3. Performance Optimierung

Performance Optimierung ist nicht Bestandteil dieser Arbeit. Da die Lastkontrolle die Anfragen der Applikation zeitweise blockiert, wird die Performance für die Applikation selbst sogar sinken. Jedoch können durch die Lastkontrolle Freiräume für andere Anwendungen garantiert werden und so die „gefühlte“ Leistung des Gesamtsystems für den Anwender steigen. Dies ist jedoch keine Optimierung der Performance im engeren Sinne.

Teil II.

Ressourcen und Lastkontrollen

3. Ressourcen, Metriken und Last

Ein Grundbaustein dieser Arbeit ist der Umgang mit *Ressourcen* (z.B. der CPU oder dem Speicher eines Systems) und *(Software-) Metriken*[1], die diese Ressourcen unter *Last* messbar machen. Im Folgenden werden diese Begriffe eingeführt und anhand von Definitionen eingegrenzt. Außerdem wird beschrieben, welche Metriken für die hier vorgestellte Anwendung von besonderer Relevanz sind und daher exemplarisch herangezogen werden sollen.

3.1. Definitionen

3.1.1. Ressourcen

Ressourcen (auch *Betriebsmittel* genannt) „sind alle Komponenten eines Computersystems, die zur Erfüllung von Aufträgen benötigt werden" (Schneider and Werner, 2007, Seit 277). Dies schließt sowohl Software und Daten als auch Hardware (z.B. die CPU) mit ein. Man unterscheidet Ressourcen anhand verschiedener Kriterien. Hierzu gehört etwa die *Wiederverwendbarkeit*[2] und die *Benutzungsweise*[3].

Im Allgemeinen werden Ressourcen vom Betriebssystem verwaltet. Dieses hält die Kontrolle über die Belegung der Ressourcen und weist diese nach einem bestimmten Verfahren den einzelnen Prozessen zu. Man spricht hierbei vom Scheduling (dt. *Ablaufplanung*).

Datenbanksysteme betreiben ebenfalls in der Regel selbst eine (Art von) Ressourcenverwaltung. Beispielsweise kontrollieren Datenbanken, den Zugriff auf Datensätze, obwohl diese Softwareressourcen eigentlich der Speicherverwaltung des Betriebssystems unterliegen. Auch betreiben solche Systeme häufig eigene Puffer und Speicherbereiche, die sie selbst verwalten.

[1] Im Folgenden auch *Kennzahlen* oder *Leistungsdaten* genannt.
[2] Signale oder Events sind beispielsweise nur einmalig verwendbar.
[3] parallel oder sequentiell

3.1.2. (Software-) Metriken

Metriken sind Kennzahlen, die zur Erfassung von Messwerten dienen. Man unterscheidet bei hierbei zwischen *absoluten*[4] und *relativen Metriken*[5].

In der Informatik spricht man im Speziellen von sogenannten Softwaremetriken - diese ordnen einem (Software-) Prozess eine Kennzahl zu. Mit ihnen ist es möglich, einen solchen Prozess quantitativ zu beschreiben und einzuordnen. „So kann durch den Metrikeinsatz die Qualität der Produkte bzw. Prozesse zu einem Zeitpunkt im Sinne einer Standortbestimmung in quantitativer Form ermittelt werden“ [Schneider and Werner, 2007, Seite 254].

3.1.3. Last

Last bezeichnet das Arbeitsaufkommen, welchem ein System ausgesetzt ist. Dieses Arbeitsaufkommen kann unterschiedlicher Art sein. Denkbare Lasten sind etwa die Anzahl der auf dem System aktiven User, die Frequenz von eingehenden Anfragen oder die Anzahl parallel laufender Prozesse.

Last kann ein wichtiger Kontext für Messungen und Tests sein. So lässt sich etwa die Antwortzeit für einen bestimmten Request nur dann genau angeben, wenn die Rahmenbedingungen (wie z.B. aktive User und laufende Prozesse) genau definiert sind - eben die Last, unter der gemessen wurde.

Dieser Zusammenhang wird auch als *Lastempfindlichkeit* (engl. *Load sensitivity*) bezeichnet (siehe dazu Fowler et al., 2002). Damit ist die Abhängigkeit von Leistungsindikatoren (z.B. der Antwortzeit) von der aktuellen Last gemeint.

3.2. Kennzahlen für diese Arbeit

Ressourcen und Metriken spielen in dieser Arbeit eine wichtige Rolle. Einerseits dienen sie dem Lastkontrollsystem dazu, Performancedaten abzufragen (sieh hierzu Kapitel 7) und diese anhand von definierten Regeln zu bewerten (siehe hierzu Kapitel 8). Andererseits werden sie auch zur Beurteilung des Lastkontrollsystems herangezogen (näheres hierzu in Kapitel 11).

[4] z.B. die Größe eines Speichers.
[5] z.B. Transaktionen pro Zeiteinheit.

Da es jedoch eine nahezu beliebige Anzahl möglicher Metriken gibt, sollen für diese Arbeit nur einige wichtige herangezogen werden. Tabelle 3.1[6] zeigt eine Reihe bedeutsamer Metriken[7]. Besonders hervorzuheben ist dabei *die prozentuale Auslastung der CPU* [8] und *die beendeten Transaktionen pro Zeiteinheit.* Diese werden in den folgenden Kapiteln beispielhaft verwendet.

Für diese beiden Metriken spricht, dass sie zum einen leicht zu messen und zum anderen bestimmend für die Performance der Datenbank sind. Die CPU Auslastung ist zudem ein genereller Indikator für die Leistung eines Systems. Außerdem ist davon auszugehen, dass beide Größen direkt voneinander abhängen, worauf in Kapitel 11 näher eingegangen werden soll.

Metrik	**Erklärung**
Prozentuale Auslastung der CPU pro Datenbank	Gibt wieder, wie stark die jeweiligen Datenbanken die CPU belasten. Die CPU-Auslastung ist ein wichtiger Indikator für die Leistung des Systems.
Beendete Transaktionen pro Zeiteinheit	Kein direkter Performance-Wert einer Ressource. Hat man aber stets mit ähnlichen Transaktionen zu tun, so kann über diesen Wert auf andere konkrete Auslastungen geschlossen werden.
Gesamtauslastung der CPU	Gibt wieder, wie stark die CPU insgesamt ausgelastet ist. Hier fallen auch andere Datenbanken und Anwendungen ins Gewicht.
Prozentuale Zeit, die die Datenbank während des Messintervalls auf die CPU warten musste	Je höher dieser Wert, desto häufiger war die CPU beschäftigt und nicht verfügbar. Ist die Zeit groß, so muss die Datenbank häufig und lange auf die CPU warten.
Gesamte Auslastung des zIIP-Prozessors[9]	Gibt wieder, wie stark der zIIP-Prozessor ausgelastet ist. Dieser Prozessor muss vom Kunden nicht pro Takt bezahlt werden.
Prozentuale Auslastung des zIIP-Prozessors pro Datenbank	Gibt wieder, wie stark der zIIP-Prozessor von der jeweiligen Datenbank ausgelastet ist. Dieser Prozessor muss vom Kunden nicht pro Takt bezahlt werden.

Tabelle 3.1.: Wichtige Leistungsindikatoren

[6] Im Anhang dieser Arbeit findet sich eine ausführlichere Darstellung der Tabelle, die die entsprechenden RMF-Bezeichner zu den Metriken enthält.
[7] Welche Metriken im konkreten Anwendungsfall wichtig sind, muss individuell entschieden werden.
[8] Bezogen auf die jeweilige Datenbank.

4. Lastkontrollen

Ziel dieser Arbeit ist, die Entwicklung eines clientseitigen Lastkontrollsystems. Doch was ist überhaupt ein „Lastkontrollsystem“ und welche Aufgaben soll es erfüllen? Was sind die Vor- bzw. Nachteile eines solches Systems? Diese Fragen sollen im Folgenden geklärt werden. Anhand einer Definition wird die Lastkontrolle eingeführt und im Weiteren näher beleuchtet. Zur besseren Anschaulichkeit wird zudem auf ein Beispiel von Lastkontrollen neben der Informatik eingegangen.

4.1. Definition

Lastkontrolle bezeichnet die Überwachung und Regulierung der Ressourcennutzung eines Systems. Hierzu werden festgelegte Kennzahlen (siehe Abschnitt 3.1.2) gemessen, anhand derer eine Entscheidung getroffen wird. Dieser Entscheidungsfindung liegt eine Sammlung von Regeln zugrunde. Die so getroffene Entscheidung kann je nach konkretem Anwendungsfall variieren. Denkbare Reaktionen sind *Warten*, *Ablehnen neuer Anfragen* oder *das Entziehen einer Ressource*.

4.2. Notwendigkeit

Lastkontrollen sind notwendig, da die Ressourcen eines Rechners physikalisch begrenzt sind. Dies macht es erforderlich, ihren Gebrauch bzw. ihre Vergabe zu organisieren. Nur so bleiben Systeme erreichbar und brechen nicht unter zu viel Last zusammen.

Zudem ermöglichen Lastkontrollmechanismen multitaskingfähige Systeme und (quasi) parallele Verarbeitung auf Computern mit nur einem Rechenkern zu implementieren oder mehreren Nutzern gleichzeitig auf einer Datenbank zu operieren. Zu viel Last kann jedoch jedes System in die Knie zwingen.

Warum ist das so? - Vereinfacht ausgedrückt erzeugt Last *Last*. Der Rechner (egal ob Desktop-PC oder Datenbankserver) geht nicht nur „in die Knie", weil er mehr schultern muss, als die Hardware vermag, sondern auch, weil er diese Last (z.B. Prozesse oder Anfragen) organisieren muss. Jeder neue Befehl von außen will nicht nur bearbeitet werden, sondern auch entgegengenommen, bewertet, in eine Warteschlange eingereiht und fortan überwacht werden. Ist der RAM ausgeschöpft, so muss häufig ein aufwändiges Swapping[1] durchgeführt werden. All dies erzeugt zunehmenden Overhead, der zur eigentlichen Last hinzukommt.

Ab einem gewissen Punkt sinkt die Performance eines Computers daher drastisch. Das System ist dann mit „sich über Wasser halten" beschäftigt und antwortet zusehends langsamer. Diese Abhängigkeit von Last und Verarbeitungszeit wird als *Lastempfindlichkeit* bezeichnet (siehe 3.1.3, sowie Fowler et al., 2002).

Diesen Punkt der Überlastung gilt es zu vermeiden. Er kann nämlich nicht nur die Antwortzeiten eines Systems so stark verlangsamen, dass es de facto nicht erreichbar ist[2], sondern es sogar ganz zum Absturz bringen[3]. Damit ist die Verfügbarkeit des Systems nicht mehr gewährleistet.

4.3. Exkurs: Lastkontrollen neben der Informatik

Neben der Informatik spielen Lastkontrollen auch in anderen Gebieten eine wichtige Rolle - wenn auch häufig unbewusst. Interessant sind hierbei die Analogien zur Softwaretechnik. Zwar hat jeder Vergleich Schwächen, doch sollen vor allem die Ähnlichkeiten zur Verdeutlichung hier herausgestellt werden.

Sicherungen regulieren z.B. die Last auf elektrische Bauteile. Die Kennzahl ist hierbei eine Stromstärke, ab der die Sicherung mit *Abschalten* reagiert. Die Sicherung ist dabei zwischengeschaltet: weder das zu schützende Bauteil, noch das Stromkraftwerk wissen von ihr. Jedoch regelt sie klar einen Kontext: mehr als eine definierte Stromstärke lässt sie nicht durch. Darauf kann sich das dahinterliegende Bauteil verlassen und muss sich nicht mehr um diese Aufgabe kümmern - ähnlich wie ein Workload hinter einer transparenten Lastkontrolle.

[1] Swapping meint das temporäre Auslagern von Daten im RAM auf die Festplatte. Dieser Vorgang ist zeitraubend, da die Festplatte um vieles langsamer ist als der RAM. Wird dieser Vorgang daher häufiger ausgeführt, verlangsamt dies die Verarbeitung erheblich.
[2] Dieses Szenario entspricht dem Praxisbeispiel des Performance Experts Teams aus der Einführung.
[3] Bestes Beispiel hierfür sind DoS-Attacken gegen Webserver.

Abbildung 4.1.: Last im Straßenverkehr (Quelle: `http://www.koeln.de`)

Im Straßenverkehr steuern Ampeln die Verkehrsdichte an kritischen Straßenabschnitten wie Tunnels oder viel befahrenen Autobahnen. Hierbei ist die Kennzahl das Verkehrsaufkommen, in Folge dessen die Strecke möglicherweise geschlossen wird, um einen Stau zu verhindern.

Hierbei ist ein „Knick" im Verlauf zwischen Verkehrsaufkommen und (z.B.) Geschwindigkeit zu erkennen. So liegt das durchschnittliche Tempo auf einer dreispurigen Autobahn etwa bei ca. 100 km/h bei 50 Fahrzeugen pro Kilometer. Schon bei etwa 75 Fahrzeugen pro Kilometer sinkt die Geschwindigkeit auf ca. 60 km/h (näher ausgeführt wird dieses Thema in Wu, 2000). Dies ist mit der Überlastung eines Rechners und einbrechenden Antwortzeiten zu vergleichen.

All diese Maßnahmen dienen dem Zweck, das Zielsystem zwar möglichst optimal auszulasten (z.B. soviel Verkehr wie möglich fließen zu lassen), aber keinesfalls zu *überlasten*, was eventuell zu einer unerwünschten Situation führen könnte - wie etwa einem Stau.

4.4. Bewertung von (clientseitigen) Lastkontrollen

Wie die meisten Techniken, so ist auch ein Lastkontrollsystem kritisch zu sehen. Neben Vorteilen beinhaltet es naturgemäß auch Nachteile. Beide Punkte werden im Folgenden erörtert. Kritisch wird dabei insbesondere die clientseitge Lastkontrolle als Gegenstand dieser Arbeit diskutiert.

4.4.1. Vorteile

Lastkontrollen können „Schlimmeres“ verhindern. Sie schützen Systeme vor Überlastung und stellen dadurch deren Funktionsweise sicher. Sie definieren einen klaren Kontext von Last, dem das System (höchstens) ausgesetzt ist. Lastkontrollen ermöglichen so, begrenzte Ressourcen besser zu nutzen - nämlich bis *fast* an ihre Grenzen. Dabei entbinden Lastkontrollen je nach Umsetzung den Client bzw. den Server von der Verantwortung selbst für eine Begrenzung der Last zu sorgen.

Wie bereits in Kapitel 2.1 erläutert, beschränkt eine Firewall etwa die Zugriffe pro Zeiteinheit auf einen Webserver. Dieser muss sich somit nicht mehr selbst um dieses Problem kümmern[4]. Tragen Datenbankanwendungen Sorge für die Zahl ihrer Statements, muss dies nicht mehr im DBMS geregelt werden. Dies entspricht dem Prinzip der *Separation of Concerns*, also der Aufteilung von Verantwortlichkeiten auf einzelne Komponenten (Eilbrecht and Strake, 2010, Seite 5 f.).

Lastkontrollsysteme können auch in einem weiteren Schritt dazu genutzt werden, um beispielsweise Logging oder Monitoring durchzuführen - sie sitzen schließlich an zentralen Schlüsselpunkten. Ein Load Balancing kann ebenfalls auf einer Lastkontrolle aufbauen.

Clientseitige Lastkontrollen bieten zudem den Vorteil, keine Last (durch sich selbst) auf dem Serversystem zu verursachen - sie spielen sich ausschließlich auf dem Client ab. Dadurch können sie auch einfacher und feingranularer eingerichtet werden. Für jede Anwendung kann eine spezielle Lastkontrolle umgesetzt werden, die genau auf diese Anwendung abgestimmt ist. Eine clientseitige Lastkontrolle kann die Anwendung kennen, die sie kontrolliert, eine serverseitige Lastkontrolle muss generell sein.

4.4.2. Nachteile

Der Betrieb einer Lastkontrolle stellt einen Aufwand dar. Sie muss eingerichtet, betreut, konfiguriert und weiterentwickelt werden. Außerdem stellt sie auch einen Rechenaufwand für den Computer selbst dar, was wiederum zu *Last* führt. Da Lastkontrollsysteme große „Macht“ besitzen, können schlecht implementierte Verfahren selbst zu Problemen wie Starvation[5] oder Locking Escalations[6] führen.

[4] Würden die Zugriffe z.B. im Falle eines DoS-Angriffes zuerst bis zum Webserver gelangen bevor sie blockiert würden, wäre es ohnehin zu spät. Schließlich wäre dann die Last schon beim Server angelangt.

[5] Starvation (engl. Verhungern) meint das Warten eines Prozess auf eine benötigte Ressource die er nie bekommt. Dies könnte auch bei Anfragen an ein Datenbanksystem der Fall sein, die vom Lastkontrollsystem niemals durchgelassen werden.

[6] Zögert eine clientseitige Lastkontrolle etwa die Ausführung eines einzelnen Statements aufgrund von Performancedaten heraus, so dauert die gesamte Transaktion länger. Da diese eventuell Locks hält, kann dies Auswirkungen auf andere Transaktionen haben, die so ebenfalls warten müssen (usw.).

Die clientseitige Lastkontrolle beinhaltet außerdem den bedeutsamen Nachteil, dass sie nur auf *einen* Client (nämlich sich selbst) Einfluss nehmen kann. Überwacht sie den Server ausreichend, so kann sie zwar über die anderen Clients Kenntnis erlangen, diese aber nicht beeinflussen. Sie agiert *nur* lokal.

Was aber nützt eine Lastkontrolle, die nur einer von vielen Clients betreibt? - Die Antwort ist simpel: nichts (siehe hierzu etwa das Beispiel unter 11.2.3.2). Daher muss ein solches System auf allen Clients gleichermaßen eingerichtet sein. Nur im Falle, dass eine Applikation bewusst „zurücksteckt“ und eine Bevorzugung der anderen in Kauf nimmt, würde eine einzelne, clientseitige Lastkontrolle Sinn ergeben.

Solche Situationen kennt man z.B. aus dem Multithreading, wo Threads mit niedriger Priorität gestartet werden, um Hintergrundaufgaben zu bearbeiten. Ihnen werden Threads mit höherer Priorität vorgezogen. Die Funktionsweise ist hierbei analog zu einer clientseitigen Lastkontrolle, nur dass die Entscheidungsgewalt umgedreht ist: bei Threads entscheidet die Laufzeitumgebung, welcher von ihnen laufen darf, die clientseitige Lastkontrolle stoppt hingegen von sich aus.

Die in dieser Arbeit entwickelte Schicht ergibt also für Applikationen Sinn, die nicht möglichst *schnell* sein müssen (z.B. Test-Workloads oder langlaufende Reportingprogramme) und daher hinter anderen Anwendungen zurückstecken können. Ist die Last hoch, so warten sie von selbst, bis die Lastentwicklung wieder günstig ist, um weiterzuarbeiten. Auch gibt die hier vorgestellte Applikation Sinn, wenn sie auf einer Reihe von Rechnern gleichermaßen eingerichtet ist. Dieses Szenario wird in Abschnitt 11.2.3 gesondert betrachtet.

Nicht zweckmäßig ist die hier vorgestellte Anwendung, wenn sie nur auf einem einzelnen System von vielen installiert wäre und dieses einen möglichst hohen Durchsatz erzielen müsste.

Teil III.

Implementierung

5. Ansatzpunkte für die Implementierung

Das zu entwickelnde System muss in der Lage sein, zwei grundsätzliche Aufgaben zu lösen: Zum Einen muss es regulierend auf die Kommunikation zwischen Client und Datenbank einwirken können. Dies ist nötig, um die Lastkontrolle (etwa durch *Pausieren*) durchsetzen zu können. Zum Anderen muss es Leistungsdaten der Datenbank bzw. des darunterliegenden Betriebssystems abfragen und auswerten können. Nur so ist es möglich, Entscheidungen zu treffen, *ob* und *wie* die Kommunikation der Teilnehmer beeinflusst wird. Wie also diese beiden Punkte umsetzen?

Im Folgenden werden verschiedene Möglichkeiten erörtert und der eingeschlagene Lösungsweg begründet. Anschließend wird in Abschnitt 5.1.3 der gewählte Ansatzpunkt, die Kommunikation zwischen Client und Datenbank zu kontrollieren, vorgestellt. Die umgesetzte Lösung für die Abfrage von Leistungsdaten wird in Abschnitt 5.2.3 gezeigt.

5.1. Möglichkeiten für transparente Zwischenschichten

5.1.1. Eine Zwischenschicht auf Netzwerkebene

Die Verbindung zwischen Client und Server ist das Netzwerk - hierüber läuft ihre *gesamte* Kommunikation. Damit eignet sich das Netzwerk als Eingriffspunkt, um diese Kommunikation zu beeinflussen. So können beispielsweise Netzwerkpakete abgefangen, verworfen, verzögert oder sogar verändert werden. Dieses Prinzip findet sich beispielsweise bei Personal Firewalls wieder, die auf Desktop-Rechnern zum Einsatz kommen.

Diese Lösung ist grundsätzlich denkbar, jedoch in mehrerer Hinsicht problematisch. Sie wäre stark an das jeweilige Betriebssystem gebunden, da mit *Low-Level*-Funktionen gearbeitet werden müsste, um die Netzwerkkommunikation zu beeinflussen. Die Schicht könnte dabei auch in Konflikte mit anderen Programmen kommen, wie etwa den eben erwähnten Personal Firewalls. Außerdem ließe eine solche Schicht nur eine begrenzte Reglementierung zu. Die Verzögerungszeit für Pakete wäre beispielsweise beschränkt, da sonst Time-Outs entstehen könnten; sowohl im Server, der auf eine Antwort wartet, als auch im Client, der den Server versucht zu erreichen.

Letztendlich würde auch die Analyse von Netzwerkpaketen einen hohen Aufwand bedeuten: Fragmentierte Pakete müssten zusammengesetzt werden, Protokolle verstanden, Inhalte ausgelesen und in verwertbare Informationen umgewandelt werden[1]. Dieses Vorgehen wäre sehr komplex und daher fehleranfällig.

Vorteilhaft an dieser Lösung ist, dass sie ohne Änderungen an Anwendungen eingesetzt werden kann. Ist sie einmal installiert, so kann sie für alle Programme genutzt werden. Dabei spielen Programmiersprache oder Laufzeitumgebung keine Rolle. Dieser Vorteil kann aber auch zum Nachteil werden: man muss auf Netzwerkebene klären, zu welchen Programmen die Pakete gehören, um gezielt nur bestimmte Programme zu beeinflussen.

In Anbetracht der Nachteile (Aufwand, Fehleranfälligkeit und Abhängigkeit vom Betriebssystem) wird diese Lösung daher verworfen.

5.1.2. Ein stellvertretender Server

Anstatt sich als Client direkt mit dem Server zu verbinden, wäre auch die Nutzung eines stellvertretenden Servers denkbar. Er könnte sowohl auf Clientseite als auch auf Serverseite implementiert bzw. betrieben werden. Dieser stellvertretende Server könnte Anfragen des Clients entgegennehmen und sich ihm gegenüber wie der „echte“ Server verhalten; dem Server gegenüber könnte er sich wie ein Client verhalten und so einen *Proxy* zwischen beiden darstellen. In ihm könnten Entscheidungen anhand von Leistungsdaten getroffen und die Kommunikation so beeinflusst werden. Der Server müsste dazu entweder im Client bewusst als Verbindung konfiguriert oder ihm untergeschoben werden, etwa durch Abändern von IP-Adressen.

Von der Zwischenschicht auf Netzwerkebene unterscheidet sich dieser Lösungsweg vor allem durch seine höhere Abstraktionsstufe: während die Netzwerklösung mit Low-Level-Funktionen arbeitet, kann der stellvertretende Server moderne Bibliotheken und Frameworks nutzen. Man muss sich bei ihm nicht mit Protokollen oder Betriebssystemdetails auseinandersetzen, sondern kann oben auf dem OSI-Model aufsetzen (siehe hierzu Schneider and Werner, 2007, Seite 302, ff.).

Trotz allem: Einen Server zu implementieren, der sich entsprechend der JDBC-Spezifikation ansprechen lässt, ist aufwändig. Besonders in Anbetracht dessen, dass nur ein kleiner Teil der Kommunikation relevant ist (z.B. die Methode `execute` eines Statements, nicht aber `close`) - jedoch muss im Server auch die nicht relevante Kommunikation abgearbeitet werden. Daher muss viel Arbeit investiert werden, um wenige Aspekte der Kommunikation zu beeinflussen. Eine genauere Lösung ist daher wünschenswert.

[1] Dies wäre z.B. im Falle von *verschlüsselter* Kommunikation gar nicht möglich.

5.1.3. Der Datenbank-Treiber

In Programmen werden Datenbankzugriffe in der Regel über einen speziellen Treiber abgewickelt. Dieser stellt eine API für Datenbanken dar, die die Hersteller mit einer Implementierung für ihr jeweiliges Produkt bedienen. Im Fall Java[2] ist dies der sogenannte JDBC-Treiber. Da alle Datenbankzugriffe einer Applikation durch ihn abgewickelt werden, kann hier angesetzt werden, um regulierend in diese Kommunikation einzugreifen. Durch ihn gehen sowohl Anfragen an die Datenbank, als auch Antworten von dieser. Für den hier vorliegenden Anwendungsfall ist jedoch nur ersteres von Bedeutung.

Der JDBC-Treiber hat verschiedene Vorteile gegenüber den zuvor vorgestellten Lösungen. Er ist rein in Java geschrieben und stellt somit eine abstrahierte Sicht auf die Kommunikation dar. Mit dieser Sicht kann einfach gearbeitet werden, z.B. wesentlich einfacher als mit Netzwerkpaketen. Außerdem hat der JDBC-Treiber einen beschränkten Umfang und ist klar strukturiert. Einzelne Klassen und Methoden erfüllen gut dokumentierte Aufgaben, auf die dezidiert eingewirkt werden kann. Daher ist hier der nötige Aufwand - verglichen mit den anderen Lösungen - gering. Die Zwischenschicht des hier vorgestellten Lastkontrollsystems wurde daher mithilfe des JDBC-Treibers umgesetzt. In Abschnitt 6 werden drei Methoden gegenübergestellt, mit denen durch den JDBC-Treiber Einfluss auf die Kommunikation zwischen Client und Datenbank genommen werden kann.

5.2. Möglichkeiten zur Abfrage von Performance-Daten

5.2.1. Eigene Implementierung und allgemeine Frameworks

Performance-Daten lassen sich auf verschiedene Weisen abfragen. Meist bieten Betriebssysteme eigene Befehle bzw. Tools für solche Zwecke. In Linux lassen sich etwa Daten wie die CPU Auslastung über das Software-Tool `sar` auslesen (nähere z.B. unter Natarajan, 2011). Auf z/OS wäre ein Pendant das DIAG (Diagnostic) Interface, mit dem sich unter anderem auch die CPU-Auslastung abfragen lässt.

Auf Grundlage solcher Schnittstellen lässt sich eine eigene Monitoring-Software implementieren. Außerdem existieren verschiedene (teils quelloffene) Frameworks, wie z.B. SIGAR (`http://www.hyperic.com/products/sigar`), die eine API für die Abfrage von Performance-Daten auf mehreren Systemen bieten. Jedoch existieren solche Bibliotheken nicht für z/OS - lediglich für die Linux-Variante (z/LINUX), die auf SYSTEM z Hardware läuft, existieren solche Bibliotheken. Da der Hauptfokus der Anwendung aber gerade auf z/OS liegt, kommt diese Lösung nicht in Frage.

[2] Hierauf soll in dieser Arbeit der Fokus liegen.

5.2.2. DB2 z/OS Instrumentation Facility Interface

Die DB2 Datenbank auf z/OS bietet mit dem DB2 Instrumentation Facility Interface (kurz IFI) eine Option an, um Datenbank-spezifische Leistungsdaten abzufragen. Hierzu gehören etwa Ausführungszeiten von Statements oder Deadlocks.

Über das IFI lassen sich sogenannte IFCIDs aktivieren. Dies sind formalisierte Textblöcke, in denen DB2 Leistungsdaten mitschreibt. Aktuell existieren ca. 400 verschiedene IFCIDs für DB2 10 auf z/OS. Diese können nach Bedarf einzeln an- bzw. abgeschaltet werden. IFCID 316 enthält beispielsweise Informationen zu dynamischen Statements, wohingegen IFCID 233 Informationen zu ausgeführten Stored Procedures enthält (siehe IBM, 2011b, Seite 1101 bzw.1039).

Wird ein IFCID aktiviert, so schreibt DB2 dieses kontinuierlich in einen bestimmten Speicherbereich des z/OS-Systems. Dort ist es als reiner Text abgelegt der geparst und ausgewertet werden kann. Dies kann entweder auf Hostseite geschehen (z.B. durch z/OS-Sprachen wie REXX die speziell für solche Anwendungen konzipiert sind) oder auf Clientseite. Hierzu müssen die Textfiles jedoch über das Netzwerk auf den Client transportiert werden (z.B. durch FTP).

Anhand der IFI-Dateien können viele Datenbank-spezifische Metriken (z.B. Deadlocks) abgefragt werden. Diese Metriken lassen sich im DB2 genau definieren und sind sehr aktuell, da sie permanent geloggt werden. Daher beruhen auch kommerzielle Monitore für DB2 z/OS auf diesen Daten.

Problematisch an diesen Daten ist jedoch ihre Verarbeitung für die Lastkontrollschicht. Würde die Auswertung der IFCIDs auf dem Server geschehen, so müsste viel Code auf z/OS entwickelt werden (z.B. für das Parsing, die Aggregation oder eine Schnittstelle nach Außen für den Client). Jedoch soll der Fokus der Implementierung auf einer *clientseitigen* Lastkontrolle liegen. Um die Auswertung jedoch auf dem Client auszuführen, müssten die Daten über das Netzwerk kopiert werden. Dies ist wegen der Größe der IFCID-Daten problematisch. Es wäre wünschenswert, vorab auf dem Server aggregierte Leistungswerte abfragen zu können.

Das IFI beinhaltet zudem den Nachteil, *nur* DB2-spezifische Leistungsdaten bereitzustellen. Informationen über das gesamte System (wie z.B. die Auslastung der CPU) lassen sich nicht abfragen. Ein Ziel der Lastkontrolle sollte es jedoch sein, Freiräume für Tools auf z/OS beim Ausführen von Test-Workloads für die Entwickler zu gewährleisten. Somit muss die Gesamtauslastung des Systems betrachtet werden. Da aber gerade diese Daten über die systemweiten Ressourcen nicht über das IFI bezogen werden können, scheidet es für die Verwendung für die Lastkontrolle aus.

5.2.3. Resource Measurement Facility

Die IBM Resource Measurement Facility (kurz RMF) ist ein Performance-Monitor für z/OS. Sie bietet die Möglichkeit, Leistungsdaten eines solchen Großrechners einzusehen und aus Applikationen heraus abzufragen. Als Feature von z/OS wird sie mit jeder aktuellen Version des Betriebssystems verteilt (vgl. `http://www-03.ibm.com/systems/z/OS/zos/features/rmf/index.html`) und wird nach Angaben der IBM von einer Großzahl von Kunden permanent eingesetzt.

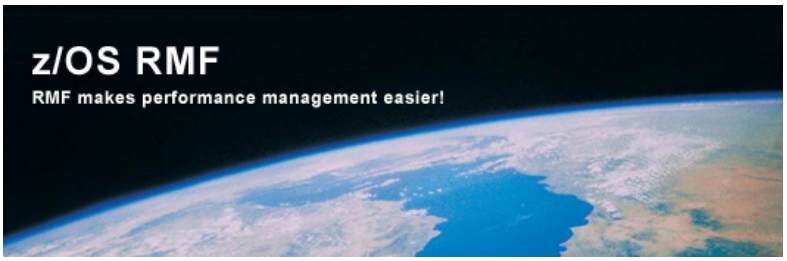


Abbildung 5.1.: Werbebanner der RMF (Quelle: `http://www.ibm.com`)

Die RMF bietet neben verschiedenen graphischen Interfaces auch eine HTTP basierte Schnittstelle zum Abfragen von Leistungsdaten. Diese liefert eine durch XSD formalisierte XML-Datei als Antwort. Der RMF-Server kann je nach benötigten Leistungsdaten konfiguriert werden. So lässt sich beispielsweise ein Intervall für die Aktualisierung der Daten nahezu[3] frei wählen. Diese Eigenschaften machen die RMF zu einer komfortablen Lösung.

Sie ist gegenüber den anderen beiden Monitoring-Varianten zu bevorzugen, da sie eine große Bandbreite an globalen z/OS -Leistungsdaten bietet. Mit ihr lassen sich alle Ressourcen von z/OS überwachen und deren Auslastung einzeln den laufenden Programmen zuordnen. Mögliche Leistungsdaten die mittels der RMF abgefragt werden können, sind etwa die *Auslastung der CPU* oder die *Transaktionen pro Zeiteinheit*. Eine Liste mit den wichtigsten Leistungsdaten findet sich in Tabelle A.

Zudem muss durch ihre Nutzung kein Code auf dem z/OS-System entwickelt werden. Damit lässt sich die gesamte Lastkontrolle rein in Java auf dem Client implementiert.

Abschnitt 7 zeigt, wie mit Hilfe der RMF aktuelle Performance-Daten für die Lastkontrolle bezogen werden können. Es wird erklärt, wie die RMF funktioniert und wie deren Daten abzufragen sind.

[3] Das minimale Intervall beträgt 10 Sekunden.

6. Eine transparente Zwischenschicht durch den JDBC-Treiber

Eine der Kernanforderungen ist, ein Lastkontrollsystem zu entwickeln, welches *ohne Änderungen am Client* genutzt werden kann. Es soll also ein bestehendes Programm als Vorgabe genutzt werden und zwingend *unverändert* bleiben. Dies hat den Hintergrund, dass die zu entwickelnde Anwendung auch für bestehende Tools (z.B. den zahlreichen Test-Workloads des Performance Expert Teams) genutzt werden soll. Während der Entwicklung war dies exemplarisch ein IBM internes Java Programm zur Generierung von SQL-Workload - der sogenannte DB2WorkloadService[1].

Um Java Code für ein Programm transparent (also unbemerkt) hinzuzufügen, gibt es mehrere Optionen. Der Schlüssel für diese Möglichkeiten ist der Mechanismus, mit dem in Java Bytecode abgelegt und geladen wird - zur Laufzeit über den sogenannten Classpath.

6.1. Schreiben eines eigenen JDBC-Treibers

Java-Bytecode ist in Class-Dateien (`*.class`) organisiert, die wiederum in Archiven (z.B. JAR-Archiven, `*.jar`) zusammengefasst werden können. Diese Archive kapseln Module bzw. ganze Programme oder Bibliotheken und stehen den Anwendungen über den Classpath zur Verfügung. Dies ist eine editierbare Liste von Class- und Archiv-Dateien, die zur Ausführungszeit nach den benötigten Klassen durchsucht wird.

Dieser Mechanismus ermöglicht, Bibliotheken auszutauschen und zu ersetzen; entweder durch das Überschreiben einer bestehenden Bibliothek oder das Abändern des Classpath. Ein möglicher Weg, eigenen Code zwischen Client und Datenbank zu platzieren, besteht darin, einen eigenen JDBC-Treiber zu schreiben. Dieser müsste die Vorgaben der JDBC-Spezifikation umsetzen und dann im Anwendungsprogramm (statt dem originären Treiber) genutzt werden.

[1] Dieses Programm war Gegenstand meines Praxissemesters und Praxisberichts, weshalb ich mich gut in dessen Aufbau auskannte. Näheres hier zu in Uhrig, 2011.

Der geschriebene Treiber kann seinerseits auf das Original zurückgreifen, um mit dessen Hilfe die Datenbankkommunikation abzuwickeln[2]. Der eigene Treiber stellt somit einen Proxy (Eilbrecht and Strake, 2010, Seite 75) zwischen der Applikation und dem originären Treiber dar.

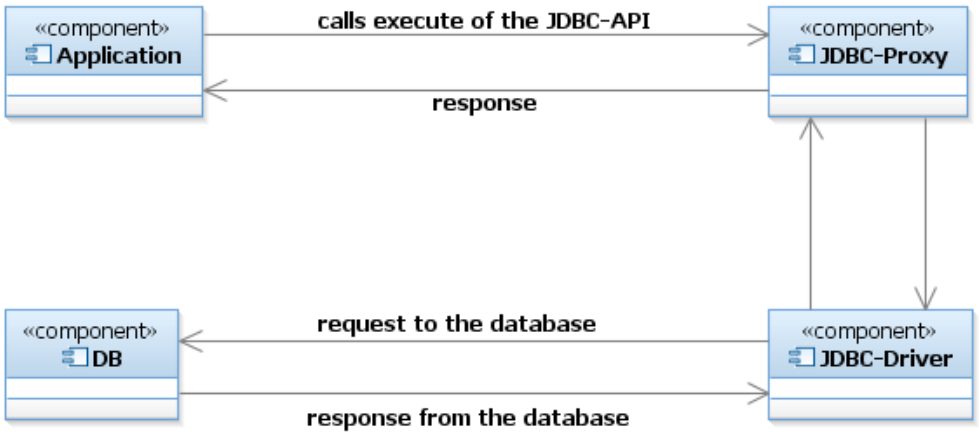


Abbildung 6.1.: Eigener JDBC-Treiber als Proxy

Hätte der Proxy-JDBC-Treiber einen eigenen Namen (der sich von dem des Originals also unterscheidet), so müsste jedoch eine Änderung im Classpath des Clients vorgenommen werden. Im einfachsten Fall zwar nur Einstellungen (z.B. in Properties-Dateien), aber trotz allem eine Änderung.

Würde die eigene JDBC-Implementierung hingegen den selben Namen tragen, wäre eine solche Änderung nicht nötig. Der Treiber könnte in diesem Fall einfach ersetzt werden. Nicht ganz so einfach gestaltet sich dann jedoch die Nutzung des Originals (da dieses schließlich einen identischen Namen trägt). Die Nutzung wäre nur auf dem Umweg eines eigenen Classloaders möglich. Dieser könnte die Originalklasse laden (da zur Identität einer Klasse auch deren Classloader gehört), die nun wiederum auf ein gemeinsames Interface gecastet werden kann. Sie auf ihren „echten" Typ zu casten, würde wegen der Namensgleichheit nicht gehen.

Algorithmus 6.1 zeigt ein exemplarisches Codebeispiel, welches eine gleichnamige Klasse in einer anderen nutzt.

[2]Die Abwicklung der Datenbankkommunikation selbst zu entwickeln, wäre nicht möglich ohne detaillierte Kenntnisse der Datenbankinterna und *viel* Aufwand. Diese Lösung ist von vorneherein auszuschließen.

Algorithmus 6.1 Nutzung einer gleichnamigen Klasse in Java

```
1  package lib.a;
2
3  public class ClassB {
4
5      public void useOriginalClass() {
6
7          try {
8
9              // zuerst wird die Lib mit einem eigenen ClassLoader geladen
10             JarClassLoader loader = new JarClassLoader("/libs/LibA.jar");
11
12             // nun kann man die benötigte Klasse finden
13             Class<?> realClassB = loader.findClass("lib.a.ClassB");
14
15             // Objekte dieser kann man auf ein gemeinsames Interface casten
16             AInterface a = (AInterface) realClassB.newInstance();
17
18             // nun lässt sich bequem mit der Klasse arbeiten
19             a.doSomething();
20         }
21         catch(Exception e) {
22             //...
23         }
24 //...
```

Problematisch an diesem Ansatz ist seine Komplexität, die - abgesehen vom bloßen Aufwand - leicht zu Fehlern führen kann. Im einfachsten Fall müsste nämlich schon die gesamte JDBC-API implementiert werden. Das sind 22 Interfaces, alleine aus dem Package `java.sql`, sowie weitere 12 Interfaces im Package `javax.sql`. Zudem kann nicht davon ausgegangen werden, dass in einer etwaigen Applikation lediglich Interfaces der offiziellen JDBC-API verwendet werden. Es könnten auch spezielle Interfaces des Treibers genutzt werden, um die Anwendung stärker an die spezifische Datenbank anzupassen. Im Falle des JDBC-Treibers von DB2 (dem sogenannten *JCC-Treiber*) könnten dies z.B. die Interfaces `DB2Statement` (im Gegensatz zum „normalen“ Statement) oder `DB2ResultSet` (analog zum „normalen“ ResultSet) sein. Dies erhöht die Zahl der zu implementierenden Interfaces drastisch - der JCC-Treiber umfasst selbst über 40 Interfaces bzw. abstrakte Klassen, *zusätzlich* zur offiziellen API.

Unter der Annahme, dass nur auf wenige Methoden dieser Interfaces überhaupt Einfluss genommen werden soll[3], wären die Mehrzahl der zu erstellenden Klassen ohnehin reine Proxies. Sie würden die Methodenaufrufe eins zu eins an die originäre Klasse weiterleiten und keinen weiteren Zweck erfüllen.

Da also alles in allem viele Klassen nötig wären, die meist jedoch keinen weiteren Nutzen haben, kommt diese Lösung nicht in Frage. Außerdem wäre sie nicht auf einen anderen JDBC-Treiber anwendbar. Dies ist zwar nicht Teil der Problemstellung, jedoch bieten die anderen Lösungen diesen Vorteil - ohne weiteren Mehraufwand spendieren zu müssen. Zudem erweisen sich andere Lösungsansätze als wesentlich punktgenauer, wie im folgenden dargelegt wird.

[3] Interessant ist z.B. die Methode `execute` des Interface Statement, kaum aber die Methode `close`.

6.2. Überschreiben einzelner Klassen des JDBC-Treibers

Der Mechanismus des Classpath ermöglicht jedoch nicht nur die Option, komplette Bibliotheken zu ersetzen, sondern auch dediziert einzelne Klassen. Diese werden in Java durch ihre Namen (inklusive Package-Namen) identifiziert[4]. Somit ist eine Klasse `package.A` aus dem Archiv `A.jar` zur Ausführungszeit identisch mit der Klasse `package.A` aus dem Archiv `B.jar`. Entscheidend ist ihre Reihenfolge im Classpath - die Klasse, die zuerst gefunden wird, wird verwendet.

Somit stellt sich die Frage: **Warum nicht nur einzelne Klassen des JDBC-Treibers ersetzen?** Dieser Ansatz würde die Zahl der zu implementierenden Klassen deutlich senken und auf diejenigen beschränken, auf die auch tatsächlich Einfluss genommen werden soll. Dieser Ansatz ist durchaus valide. Jedoch kommt es in der Praxis zu Problemen.

Im Falle des JCC-Treibers ist die an den Kunden ausgelieferte Datei *obfuscated*. Dies bedeutet, dass die Klarnamen der Klassen und Methoden durch Zufallsnamen ersetzt wurden. Obfuscating soll z.B. Software-Reengeneering verhindern - aber eben auch Anwendungen wie die hier vorgestellte. Schließlich müssten nicht die Interfaces ersetzt werden (diese sind weiterhin im Klarnamen vorhanden), sondern die „echten“ Klassen, die sie umsetzten.

Für diese Arbeit hätte prinzipiell ein Treiber, der nicht-*obfuscated* ist, zur Verfügung gestanden, welcher zumindest dieses Problem gelöst hätte[5]. Jedoch wäre dies keine praktikable Lösung gewesen, da die Voraussetzung, das Programm zu nutzen, immer dieser eigentlich *nicht zugängliche* Treiber gewesen wäre. Selbst Firmen-intern wäre die Nutzung problematisch gewesen.

Das Problem der *obfuscated*-Namen ist aber keineswegs das ausschlaggebende Kriterium - es ist ja sogar wie zuvor erwähnt lösbar. Kaum lösbar sind aber öffentliche Variablen von Klassen und Objekten. Dies ist nicht nur ein schlechter Programmierstil, sondern macht auch diese Lösung unmöglich. Das Problem soll anhand eines Bytecodeauszugs der Klasse `com.ibm.DB2.jcc.am.Statement` des JCC-Treibers deutlich gemacht werden (siehe Listing 6.2).

[4] Zur vollständigen Identität einer Klasse gehört außerdem der Classloader welcher sie geladen hat.

[5] Es sei angemerkt, dass auch dieser Treiber nur als Bytecode vorlag und keine Sourcedateien zur Verfügung standen.

Algorithmus 6.2 Bytecodeauszug der Klasse `com.ibm.DB2 .jcc.am.Statement`

```
1  // Compiled from Statement.java (version 1.2 : 46.0, super bit)
2  public class com.ibm.DB2  .jcc.am.Statement implements /*...*/ {
3
4  java.lang.String originalSql_;
5
6  // ...
7
8  public com.ibm.DB2.jcc.am.MaterialStatement materialStatement_;
9  public com.ibm.DB2.jcc.am.Connection connection_;
10 public java.sql.SQLWarning warnings_;
11 public com.ibm.DB2.jcc.SQLJSection section_;
12 public com.ibm.DB2.jcc.am.Agent agent_;
13 public com.ibm.DB2.jcc.am.ResultSet resultSet_;
14 int updateCount_;
15 int returnValueFromProcedure_;
16
17 //...
```

Wie in Listing 6.2 zu sehen, hat die Klasse mehrere öffentliche bzw. package-lokale Variablen. Auf diese kann theoretisch jederzeit von überall (bzw. aus dem Package heraus) zugegriffen werden. Eine Klasse, welche diese Klasse ersetzen soll, muss also auch diese Variablen anbieten (eine andere Klasse des JCC-Treiber könnte sie schließlich erwarten). Sie würden dann folglich in der neuen Klasse genutzt werden. Wann und von wem - dies ist anhand des Byte-Codes nicht nachzuvollziehen.

Will diese (eigene) Klasse aber wiederum das Original nutzen, so müsste sie dort ebenfalls diese Variablen setzen - sie könnten benötigt werden (dies ist anhand des Bytecodes ebenfalls nicht nachzuvollziehen). All diese Umstände wären noch zu handhaben - wenn auch in sehr unschöner und fehleranfälliger Art und Weise. So könnte etwa vor jedem Methodenaufruf der originären Klasse, deren Variablen auf die Werte der eingeschleusten Klasse gesetzt werden.

Nicht lösbar ist jedoch folgender Umstand: Ruft die originäre Klasse während der Abarbeitung einer Methode eine andere Klasse auf und setzt diese Klasse Werte in die angesprochenen, öffentlichen Felder, so tut sie dies bei der neuen Klasse, nicht dem Original. Dieses ist schließlich nicht sichtbar und wird vertreten. In der weiteren Abarbeitung der Methode kann die originäre Klasse aber sehr wohl auf diesen vermeintlich gesetzten Wert zurückgreifen und so einen Fehler verursachen (er wurde nämlich nicht bei ihr gesetzt).

Diese Situationen sind sehr schwer zu durchschauen, sodass die Vorgänge im Treiber genau analysiert werden müssten. Aber alleine die (sehr wahrscheinliche) Möglichkeit, dass auf diese Weise ein Fehler entsteht, macht diese Lösung unbrauchbar.

6.3. Instrumentierung

Die Sprache Java macht jedoch noch eine andere Lösung möglich, die - wie im folgenden beschrieben - zum Ziel führt. Benötigte Klassen einer Java-Anwendung werden nämlich erst zur *Laufzeit* auf dem Classpath gesucht und geladen. Dies ermöglicht, die Klassen durch das eigene Programm zu modifizieren - mittels **Java-Instrumentation**.

Java-Instrumentierung gestattet zum Zeitpunkt des Ladens einer Klasse, dediziert Code in dieser zu ändern. So kann eine bestehende Class-Datei abgeändert werden, beispielsweise für Monitoring Tools, Profiling oder zum Logging (siehe Lorimer, 2005). Im Falle dieser Arbeit wurde die Java-Instrumentierung dazu genutzt, in die benötigten Klassen des JCC-Treibers Kontrollstatements einzustreuen - und so eine Lastkontrolle zu ermöglichen.

Listing 6.3 zeigt einen Auszug des Agenten, also der Klasse, die beim Laden einer Class-Datei aufgerufen wird und die Bytecode-Manipulation vornimmt.

Der Mechanismus funktioniert für den Entwickler wie folgt: Zuerst wird eine Klasse mit einer sogenannten `premain`-Methode geschrieben und in der Manifestdatei für das JAR-Archiv vermerkt. Diese Methode wird noch vor der eigentlichen `main`-Methode ausgeführt und bietet die Möglichkeit, einen Agenten hinzuzufügen. Dieser erhält den Bytecode einer jeden Klasse, bevor sie „richtig“ geladen wird. Diesen Bytecode kann er beliebig manipulieren. Dies geschieht, wie in Listing 6.3 zu sehen, in der Methode `transform`, welche das Interface `ClassFileTransformer` aus dem Package `java.lang.instrument` vorgibt.

Die Manipulation kann beliebig gestaltet werden. Für die Lastkontrolle ist es vor allem wichtig, Code *vor* dem Ausführen der Methode zu platzieren. Zum Logging wird zudem Code *am Ende* der Methode eingefügt[6].

Für die eigentliche Bytecode-Manipulation wurde die **Javassist-Bibliothek von JBoss** verwendet (näheres unter `http://www.jboss.org/javassist`). Sie bietet Methoden, um Code, der als normaler String vorliegt, in den Bytecode einzufügen (Sosnoski, 2003).

[6] Dies war beispielsweise nötig, um mittels zweier Zeitstempel die Ausführungszeit von einem SQL-Kommando zu erhalten.

Algorithmus 6.3 Codeausschnitt des Agenten zur Java Instrumentierung

```
1  public class Agent implements ClassFileTransformer {
2
3      //...
4
5      public static void premain(/* Parameter */) throws /*...*/ {
6
7          //...
8
9          instrumentation.addTransformer(new Agent());
10     }
11
12     public byte[] transform(/* Parameter */) throws /*...*/ {
13
14         ClassPool pool = ClassPool.getDefault();
15
16         CtClass cl = null;
17
18         try {
19
20             cl = pool.makeClass(new ByteArrayInputStream(classfileBuffer));
21
22             for(ExecutableRule rule: rules) {
23
24                 if (/* nur Methoden für die Regeln verfügbar sind */) {
25
26                     CtBehavior[] methods = cl.getDeclaredBehaviors();
27
28                     for(CtBehavior method: methods) {
29
30                         if (/* nur Methoden die noch nicht injected wurden */) {
31
32                             injectMethod(method);
33                         }
34                     }
35
36                     return cl.toBytecode();
37                 }
38             }
39         }
40         /* catch- und finally-Blöcke */
41
42         return classfileBuffer;
43     }
44
45     private void injectMethod(CtBehavior method) throws /*...*/ {
46
47         //...
48
49         method.insertBefore(/* Code der zu Beginn ausgeführt wird */);
50         method.insertAfter(/* Code der vor jedem return ausgeführt wird */);
51     }
52 //...
```

Dieser Lösungsweg hat den Vorteil, dass nur an gewünschten Stellen mit wenig Code eingegriffen wird. Der Aufwand bleibt so minimal. Zudem ist die Lösung sehr flexibel. Wie in Listing 6.3 dargestellt, wird etwa nur Code in Methoden eingefügt, für die Regeln definiert sind (näheres zum Regelmechanismus in Abschnitt 8). Im Umkehrschluss heißt dies auch, dass wenn der Nutzer eine neue Regel definiert, auch die passende Methode instrumentiert wird. So muss nicht von Beginn an klar sein, welche Methoden von Bedeutung sind. Vielmehr bleibt dies dem Nutzer und seinen Regeldefinitionen überlassen.

Dieser Teil der Applikation ist daher unabhängig vom eingesetzten JDBC-Treiber. Es könnten sogar Klassen der eigentlichen Applikation verändert werden - falls genügend Wissen über diese vorliegt und es sinnvoll erscheint.

Einziger Nachteil dieser Lösung ist, dass in den Aufruf der Applikationen die JVM-Option für den Agenten aufgenommen werden muss. Dies ist in Listing 6.4 dargestellt. Jedoch ist dieser Eingriff denkbar klein[7].

Algorithmus 6.4 Aufruf einer Applikation mit dem Agenten des Lastkontrollsystems

```
java -javaagent:com.ibm.jdbc.load.control.jar -jar SomeApplication.jar
```

6.4. Tabellarische Gegenüberstellung der drei Varianten

Tabelle 6.1 zeigt eine Gegenüberstellung der drei Varianten. Sämtliche Möglichkeiten haben sowohl Vor- als auch Nachteile. Jedoch überwiegen bei der Instrumentierung eindeutig deren Vorteile, besonders im direkten Vergleich mit den beiden anderen Alternativen.

[7] Man könnte diesen Umstand sogar als Sicherheitsaspekt sehen: Da keine Klassen „unbemerkt“ in den Pfad eingeschmuggelt werden, sondern der Nutzer bewusst den Agenten starten muss, ist klar, dass die Lastkontrolle mit ausgeführt wird. Dies schafft mehr Klarheit - auch für Tests und Endanwender.

	eigener JDBC-Treiber	**ersetzen einzelner Klassen**	**Instrumentierung**
Sind Änderungen am Client nötig?	Falls der Treiber einen neuen Namen trägt, müssen Settings geändert werden.	Der Classpath muss angepasst werden, damit die Klasse genutzt wird.	Im Aufruf der Applikation muss der Agent mit gestartet werden.
Ist die Lösung auf verschiedene Treiber anwendbar?	Nein, da jeder Treiber eigene, spezielle Interfaces bietet, die ersetzt werden müssten.	Nein, da jeder Treiber eigene Klassen- und Packagenamen hat, müssten die Namen der Klassen angepasst werden.	Ja, es können sogar Klassen aus der Applikation injected werden, nicht nur aus dem Treiber.
Ist die Lösung fehleranfällig?	Einerseits muss viel Code geschrieben werden, wodurch auch viele Fehler entstehen können. Andererseits ist der Code meist einfach und könnte sogar generiert werden.	Ja, da die Abläufe im Treiber anhand des Byte-Codes unklar bleiben (z.B. wann öffentliche Variablen gesetzt und gelesen werden).	Der Code zur Instrumentierung ist zwar komplex, kann aber gut getestet werden und funktioniert dann für alle Treiber.
Muss von Beginn an klar sein, welche Methoden beeinflusst werden sollen?	Nein. Es können in alle neuen Methoden von vorneherein Kontrollstatements eingebaut werden. So hält man sich alle Möglichkeiten offen.	Ja, denn für genau diese Methoden bzw. Klassen müssen Ersetzungen eingeschleust werden.	Nein, es kann jede beliebige Methode injected werden.
Ist die Lösung für den Nutzer nachvollziehbar?	Ja, da der Nutzer explizit den neuen Treiber angeben muss. Es kann aber zu Verwechslungen kommen, wenn der Name der Treiber identisch ist.	Nein, da schnell der Überblick über den Classpath verloren gehen kann. Welche Klassen sind an welcher Position? Welche Klassen sind in einem JAR-File?	Ja, da der User den Agenten explizit starten muss und in den Regeln selbst angibt, welche Klassen und Methoden injected werden sollen.

Tabelle 6.1.: Gegenüberstellung der drei Varianten

7. Abfragen von Performance-Daten mittels der RMF

7.1. Grundlegendes

Die RMF ist ein **Performance-Monitor für z/OS-Leistungsdaten**. Sie läuft serverseitig und besteht aus drei Prozessen: (1) Der RMF-Hauptprozess `RMF`, (2) der Prozess zum Sammeln von Daten `RMFGAT` (vom Englischen *Gatherer*, dt. Sammler) und (3) dem Prozess `GPMSERV`. Dieser Prozess stellt unter anderem eine HTTP basierte API für den Zugriff auf die gesammelten Daten bereit.

Die RMF bietet drei Arten von Leistungsdaten an. Der sogenannte *Monitor I* sammelt Daten für Langzeitauswertungen, in Intervallen von 15 bis 30 Minuten. *Monitor II* zeigt eine Momentaufnahme von Leistungswerten an. *Monitor III* ist eine Art Kombination der beiden ersten Monitore. Er präsentiert aggregierte Werte aus kurzen bis mittleren Intervallen. Typische Intervalle für ihn sind 10 Sekunden (der kürzest mögliche Wert) bis ca. eine Minute (vgl. Cassier et al., 2005). Der Prozess `GPMSERV` stellt Daten dieses Monitors durch das HTTP-Interface zur Verfügung.

Neben den serverseitigen Komponenten bietet die RMF auch einen Webclient und eine Software für den Desktop: RMF Performance Monitoring Java Technology Edition (kurz RMF PM). Diese Java-Anwendung für Arbeitsplatzrechner ermöglicht, Leistungsdaten mit einer graphischen Oberfläche zu überwachen. Das Programm ist sowohl für Windows, als auch für Linux verfügbar und basiert auf den Daten des *Monitor III*. Abbildung 7.1 zeigt ein Screenshot dieser Anwendung.

RMF PM ermöglicht es eine individuell zusammengestellte Sammlung von Leistungswerten grafisch zu überwachen. Hierzu lassen sich beliebig viele Leistungswerte zu einem sogenannten *Perf Desk* hinzufügen, der sich mit wenigen Klicks öffnen lässt. Eine Reihe solcher gespeicherten Ansichten findet sich in Abbildung 7.1 auf der linken Seite.

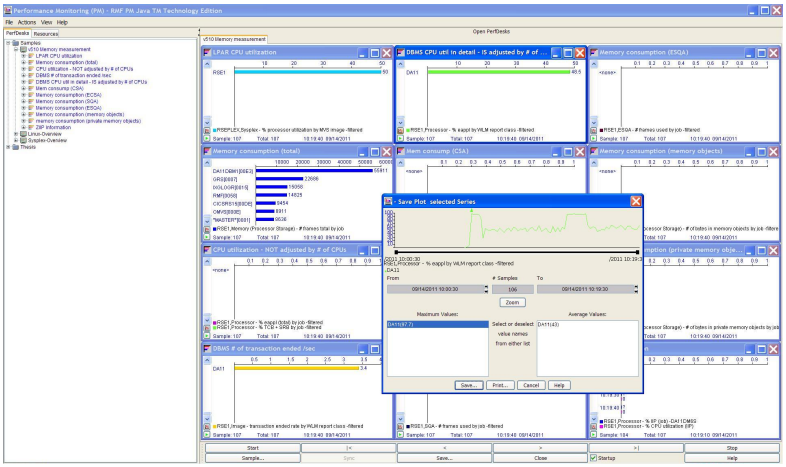


Abbildung 7.1.: Ansicht der RMF Performance Monitoring Java Technology Edition

Zudem erlaubt RMF PM die Aufzeichnung von Performance Daten über eine Zeitspanne hinweg. So lassen sich die Verläufe der Leistungsdaten verfolgen. Die aufgezeichneten Daten können als CSV-Dateien exportiert werden. Abbildung 7.2 zeigt den Verlauf der CPU-Auslastung durch eine Datenbank (*DA11*) über einen Zeitraum von ca. 20 Minuten in RMF PM.

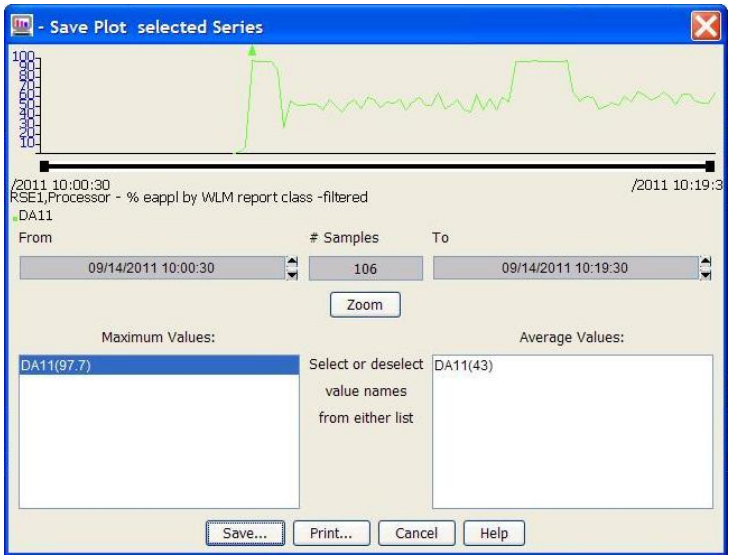


Abbildung 7.2.: Verlauf der CPU-Auslastung durch eine Datenbank in RMF PM

Einen Überblick der Architektur eines z/OS-Systems zeigt Abbildung 7.3. Dargestellt sind zwei Hardware-Boxen (*System-Z Hardware 1* und *System-Z Hardware 2*), auf denen mehrere LPARs[1] laufen. Jede LPAR hat ein eigenes Betriebssystem und ist völlig von den anderen isoliert. Den LPARs können individuell Ressourcen zugewiesen werden (z.B. CPU oder Speicher), die sie alleine oder gemeinsam nutzen können. Jedoch sind auch bei gemeinsam genutzten Ressourcen die LPARs weiterhin voneinander getrennt.

Mehrere LPARs können über eine sogenannte *Coupling Facility* zu einem *Sysplex* zusammen geschlossen werden. Sie können sich dann gemeinsame Daten teilen und so für mehr Ausfallsicherheit garantieren.

Prozesse die auf z/OS laufen werden *Jobs* bzw. *Tasks* genannt. Mehrere logisch zusammengehörige Prozesse können in einer *WLM-Class* (*Workload Manager*) zusammen gefasst werden.

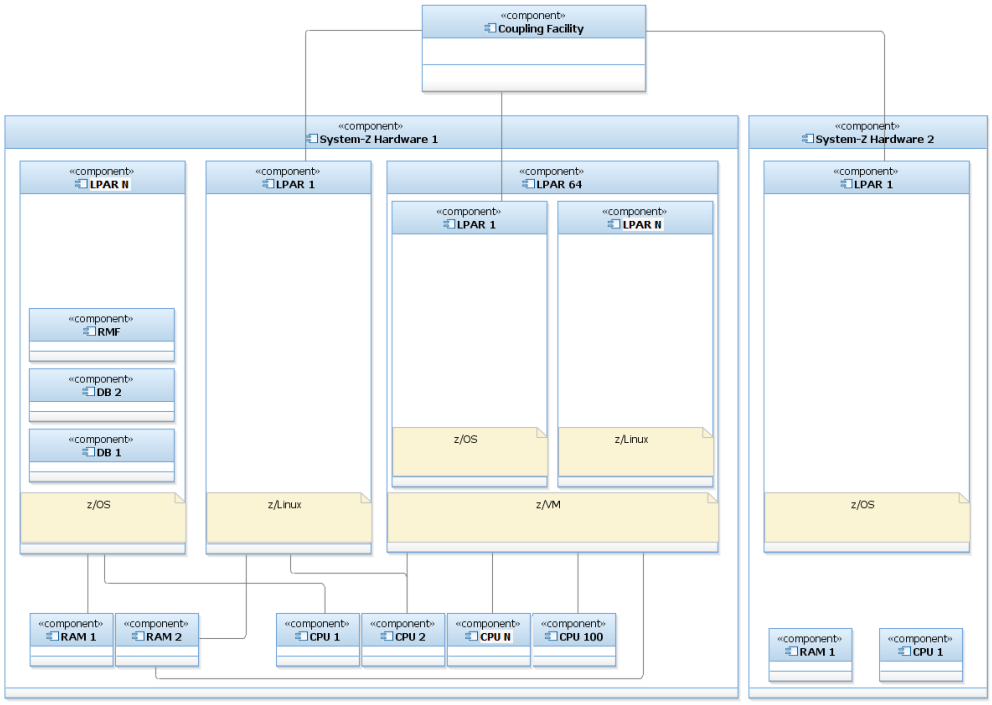


Abbildung 7.3.: z/OS-Architektur mit der RMF im Überblick

[1] Logische Partitionen

Die RMF bietet Performance-Daten für ein gesamtes Sysplex an. Die Daten sind hierarchisch nach Gruppen sortiert. So finden sich Daten zum Sysplex, Daten zu den einzelnen Maschinen, bis hin zu Daten zu den jeweiligen Jobs und WLM-Klassen. Abbildung 7.4 zeigt diese Hierarchie in einem Menu von RMF PM. Unter dem obersten Reiter „IBM z/OS“ ist der Reiter für das Sysplex aufgeklappt: „RSEPLEX,Sysplex“, wobei RSEPLEX der Name des Sysplex ist. Darunter findet sich ein weiterer Reiter für das LPAR RSE1. Hierunter sind die jeweiligen „Themengebiete“ für die Performance-Daten zu finden. (Eine ähnliche Darstellung findet sich auch als Diagramm in IBM, 2011a, Seite 45.)

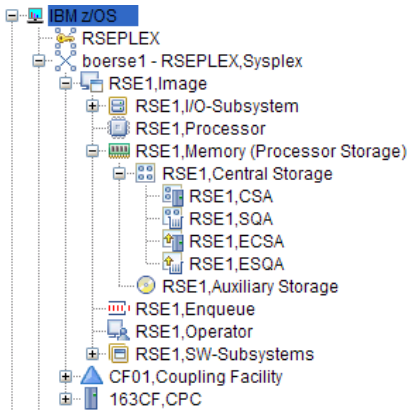


Abbildung 7.4.: Ressourcenhierarchie in RMF PM

7.2. Abfragen von Daten durch die HTTP-API

Die RMF bietet eine HTTP-API, basierend auf den Daten des *Monitor III* für Anwendungsentwickler. Diese liefert zu einem HTTP-Request (einem einfachen Aufruf einer URL) Leistungsdaten als XML. Die Syntax des XML-Formats ist durch eine XSD-Datei (`ddsml.xsd`) gegeben, die sich auf dem jeweiligen Server abrufen lässt.

Eine HTTP-Anfrage an das RMF-System besteht aus drei Teilen. Im ersten finden sich Angaben zum Server, wie Host und Port. Danach folgt eine File-Angabe. Mit ihr ist es möglich, nicht nur Leistungsdaten, sondern auch Meta-Daten abzufragen (beispielsweise, welche Metriken überhaupt verfügbar sind). Für die entwickelte Anwendung ist jedoch nur der Bezeichner `perform.xml` von Bedeutung, der die aktuellen Leistungsdaten zurückliefert. Anschließend muss dem Request eine Parameterliste mitgegeben werden, die die eigentliche Abfrage darstellt. Sie besteht aus einem `resource` Parameter, der die z/OS Ressource angibt (z.B. den Prozessor des Sysplex, in der Form `Parent,Name,Typ`), sowie einer `id`, die die benötigte Metrik referenziert (z.B. die Auslastung des Prozessors, bezogen auf eine dedizierte Datenbank). Die genauen Bezeichnungen lassen sich beispielsweise über den Webclient herausfinden. Dieser visualisiert die Daten der HTTP-API durch die Verwendung von Stylesheets.

Listing 7.1 zeigt die Request-Syntax für die HTTP-API.

Algorithmus 7.1 RMF Request Syntax (aus IBM, 2011a, Seite 48)

```
http://<host>:<port>/gpm/<file>?<parm1>=<value1>&...&<parmN>=<valueN>
```

7.2.1. Einlesen der Daten und konvertieren in Java Objekte

Um die Daten der RMF in der Applikation verwenden zu können, müssen sie zunächst eingelesen werden. In einem zweiten Schritt werden sie von ihrem XML-Format zur besseren Verarbeitung in Java-Objekte überführt.

Das Abfragen und Einlesen der Daten lässt sich über Klassen des JDK-Packages `java.net` bewerkstelligen. Zuerst wird eine URL zur betreffenden „Webseite" erstellt und zu dieser eine Verbindung geöffnet (`url.openConnection()`). Von dieser Verbindung kann wiederum ein `InputStream` Objekt bezogen werden, um den Inhalt der Webseite auszulesen - die Performance Daten.

Die Konvertierung des XML-Inputs in Java-Objekte ist mittels der Java Architecture for XML Binding (kurz JAXB) bewerkstelligt worden (näheres in IBM, 2008). Diese Bibliothek bietet unter anderem Möglichkeit, aus XSD-Dateien entsprechende Java-Klassen automatisch zu erstellen und danach anhand einer konkreten XML-Datei Java-Objekte aus diesen Klassen zu generieren (Ullenboom, 2011, Kapitel 16.4). Ein entsprechender Compiler findet sich etwa im IBM Websphere Server.

Dieses Vorgehen hat den Vorteil, für das XML-Handling nicht selbst verantwortlich zu sein. Die JAXB-API übernimmt diese Aufgabe in einer konsistenten Weise und vermeidet so Fehler.

Für die Konvertierung der XSD-Dateien in Java-Klassen bietet sich zudem eine Automatisierung (z.B. durch ANT) an. Dies hat den Vorteil, dass bei einer Änderung des XML-Schemas schnell und unkompliziert die entsprechenden Class-Dateien aktualisiert werden können. Abbildung 7.2 zeigt ausschnittsweise ein solches ANT-Skript[2].

Algorithmus 7.2 Ausschnitt eines ANT-Skripts zum Erzeugen von Java-Klassen

```
1  <project name="JAXBAutomation" default="ddsml">
2      <target name="ddsml">
3          <delete dir="./src/com/ibm/jdbc/load/control/rmf/xml/"/>
4          <java jar="libs/jaxb-xjc.jar" fork="true" failonerror="true">
5              <arg line="-p com.ibm.jdbc.load.control.rmf.xml -xmlschema ./ddsml.xsd"/>
6              <classpath>
7                  <pathelement location="./libs/"/>
8              </classpath>
9          </java>
10         <move todir="./src/com/">
11             <fileset dir="./com"/>
12         </move>
13 ...
```

[2]Nach der Generierung der aktualisierten Java-Klassen sind eventuell Änderungen im Code nötig, der diese Klasse verwendet.

7.2.2. Vorhalten der Daten

Für die Anwendung ist nötig, *aktuelle* Daten abzufragen, anhand derer eine Entscheidung getroffen werden kann. Hierfür gibt es drei denkbare Methoden:

1. **Möglichkeit 1:** Punktuelle Datenabfrage immer *dann,* wenn eine Entscheidung gefällt werden muss.
2. **Möglichkeit 2:** Vorhalten von Daten mit einer Verfallszeit (*Cache-Invalidation*).
3. **Möglichkeit 3:** Vorhalten von regelmäßig aktualisierten Daten.

Möglichkeit eins besticht vor allem durch ihre einfache programmtechnische Umsetzung. Es werden Daten abgefragt, wann immer sie gebraucht werden. Jedoch hat sie den entscheidenden Nachteil, viel *vermeidbaren* Overhead zu erzeugen. Jede Abfrage innerhalb eines Aktualisierungsintervalls der RMF würde das selbe Ergebnis liefern. Da voraussichtlich viele SQL-Kommandos in sehr kurzen Abständen abgesetzt werden, tritt dieser Fall vermutlich häufig ein - er wäre sogar die Regel. Würde nun immer eine erneute Abfrage der Daten erfolgen, so wären die Werte stets identisch, der zusätzliche Netzwerkverkehr aber erheblich.

In der Implementierung des Lastkontrollsystems wurde daher auf eine zentrale „Sammelstelle" für Daten zurückgegriffen. Hierzu werden die Daten der RMF abgefragt und in einer `Map` eines Singleton-Objekts hinterlegt. Von hier können sie von den Regeln abgefragt werden, ohne zusätzlichen Netzwerkverkehr zu erzeugen.

Um die Daten in der „Sammelstelle" zu aktualisieren, bietet es sich an Verfallszeiten mit den Daten zu speichern (*Möglichkeit 2*). Werden Werte gelesen deren Verfallszeit überschritten ist (z.B. könnten sie älter als das Aktualisierungsintervall der RMF sein), so werden diese Werte erneut abgefragt. Dieser Mechanismus wird auch als Cache-Invalidation bezeichnet.

Problematisch an diesem Vorgehen, ist seine Umsetzung in parallelem Programmcode. Setzen mehrere Threads SQL-Kommandos ab, so können auch mehrere Threads die abgelaufenen Daten „entdecken" und deren Aktualisierung anstoßen. Dieser Prozess muss daher synchronisiert werden, da die Daten lediglich einmal aktualisiert werden müssen. Pausiert die Anwendung zudem für längere Zeiträume, so sind die vorgehaltenen Werte zwangsläufig veraltet. Sie müssen beim erstmaligen Abfragen aktualisiert werden. Dabei könnten aber gerade Pausen der Applikation bzw. die I/O-intensiven SQL-Abfragen dafür genutzt werden, die Werte permanent im Hintergrund zu aktualisieren.

In der Implementierung des Lastkontrollsystems wurde daher auf *Möglichkeit drei* zurückgegriffen - eine zentrale „Sammelstelle" für Daten mit einem separaten Thread als „Taktgeber". Dieser Thread fragt in regelmäßigen Abständen die Daten der RMF ab. So können auch Pausen und I/O-intensive Abfragen zur Aktualisierung der Daten genutzt werden.

Da die Funktion der Applikation von der Aktualität der Leistungsdaten abhängt, ist es wichtig, diese sofort nach ihrer Aktualisierung in der RMF zu beziehen. Hierzu erfolgt zu Programmbeginn eine automatische Synchronisation des Lastkontrollsystems mit der RMF. Dies gewährleistet, dass die Performance-Daten immer kurz nach ihrer Erneuerung von der RMF abgefragt werden. Um zu garantieren, dass der Thread die Daten in immer gleichen Abständen abfragt, wurde die Klasse `Timer` des JDKs genutzt. Diese garantiert über einen längeren Zeitraum eine wiederholte Ausführung in immer gleichen Abständen (vgl. Ullenboom, 2011, Kapitel 9.13). So wird verhindert, dass die RMF und die lokale Anwendung nach der Synchronisation auseinander laufen.

Die permanente Vorhaltung von Daten bietet noch andere Vorteile. So kann hierdurch etwa ein lückenloses Logging der Daten erfolgen[3] oder es können historische Werte[4] gespeichert und aggregiert werden[5].

7.2.3. Abfrageintervalle

Ausschlaggebend für eine schnelle Reaktion auf Änderungen der Systemauslastung ist die *Aktualität* der Daten. Je neuer die Daten sind, desto geringer die „Trägheit" der Anwendung. Sind die Daten hingegen mehrere Minuten alt, so kommt auch die Reaktion mehrere Minuten zu spät - und die momentane Situation kann schon wieder eine ganz andere sein.

Was ist also ein „ideales" Abfrageintervall? - Die Antwort fällt kurz aus: Kein Intervall ist *ideal*. Es muss zwingend ein Kompromiss zwischen *Aktualität* und *Aggregation* eingegangen werden. Je kürzer das Intervall, desto aktueller sind die Daten. Je länger das Intervall, desto mehr entsprechen die aggregierten Daten[6] jedoch der realen, durchschnittlichen Auslastung. Über die Zeit werden kurze Leistungsspitzen bzw. Leistungseinbrüche im Mittel ausgeglichen[7]. Zudem gibt es Ressourcen, für die sich in zu

[3] Das Lastkontrollsystem schreibt z.B. eine CSV-Datei für jeden Request, die zur Auswertung dienen.

[4] Für historische Werte, z.B. der Änderung zwischen dem aktuelle und dem vorangegangenen Wert, darf ebenfalls keine Lücke im Verlauf entstehen. Pausiert die Applikation etwa für eine Minute, so können bei der Cache-Invalidation bis zu 6 Intervalle verloren gehen. Die Änderung zwischen dem aktuellen Wert und dem vorherigen macht dann keinen Sinn mehr.

[5] Der Layer speichert etwa die Änderung zwischen dem aktuellen Wert und dessen Vorgänger zwischen.

[6] Über das gesamte Intervall hinweg.

[7] Hier bietet sich eine Analogie aus dem Alltag an: Beim Anfahren eines Autos zeigen Verbrauchsan-

kurzen Intervallen überhaupt keine prozentuale Auslastung bestimmen lässt. So ist z.B. eine CPU immer 100% beschäftigt oder überhaupt nicht - entweder führt sie gerade eine Instruktion aus oder eben nicht. Werte dazwischen gibt es nicht. Will man aber die CPU-Auslastung auf 60% senken/steigern, so ist zwangsläufig die Auslastung im Mittel über eine größere Zeitspanne gemeint.

Die Abfrageintervalle in der Applikation entsprechen den jeweiligen Intervallen der RMF, da sämtliche Performance-Daten von dieser bezogen werden. Das Intervall kann daher nicht weniger als 10 Sekunden betragen, was aber zu durchaus brauchbaren Ergebnissen führt (siehe hierzu Kapitel 11). Je nach Anwendung (z.B. eher langlaufende als kurz laufende Statements) kann die Zeit im Rahmen der Möglichkeiten der RMF variiert werden. Das Lastkontrollsystem fragt die Intervalllänge der RMF automatisch ab und passt sich dieser an. Außerdem synchronisiert sie sich mit der RMF, um neue Daten möglichst frühzeitig abzufragen (also ohne Verzögerung).

zeigen oft Werte jenseits von 100 Liter pro Kilometer an, wobei sich der reale Benzinverbrauch im Mittel aber schon nach kurzer Zeit weit darunter bewegt. Welcher Wert spiegelt eher die Realität wieder? Welcher Wert ist für einen Entwickler interessanter? Meiner Meinung nach der im Mittel.

8. Regelmechanismus

8.1. Anforderungen

Jeder Entscheidung liegt eine Regel zugrunde, auf Basis derer die Entscheidung getroffen wurde. Eine Regel definiert eine Prämisse („Wenn die Datenbank ausgelastet ist, dann...”), sowie eine Konsequenz, die eintritt, wenn die Prämisse erfüllt ist („...werde ich 5 Minuten warten!”). Nach diesem allgemeinen Schema sollten sich auch Regeln für das Lastkontrollsystem definieren lassen. Es sollten u.a. Regel wie die folgenden möglich sein:

- ▷ Liegt die Auslastung der CPU für die Datenbank über 40 %, warte 3,5 Sekunden!
- ▷ Liegt die Auslastung der CPU für die Datenbank über oder genau bei 60 %, warte 3 Sekunden. Überprüfe dies drei mal!
- ▷ Liegt die Auslastung der CPU für die Datenbank über 40 %, warte bei jedem zweiten Statement 2 Sekunden!

Die Regeln sollten frei in einem Text-Format definierbar und auch während der Ausführungszeit der Applikation änderbar sein. So kann man Regeln im laufenden Betrieb anpassen und die Werte entsprechend korrekt einstellen. Außerdem dienen die Regeln dazu, die nötigen Methoden für die Instrumentierung zu identifizieren und Requests zur RMF zu definieren. Eine einzelne Regel soll sich dabei immer auf genau eine Methode und eine Liste von RMF-Requests beziehen. So sind detaillierte Regelsysteme möglich, mit denen punktgenau in den Ablauf der Applikation eingegriffen werden kann.

8.2. Regelformat

Als Format für die Regeln wurde XML gewählt, da diese Auszeichnungssprache sowohl gut zu verarbeiten, als auch einfach für den Nutzer zu lesen ist. Außerdem können die Regeln so mit jedem beliebigen Texteditor bearbeitet werden. Regeldateien lassen sich einfach abspeichern, kopieren und austauschen.

Zur Festlegung der Syntax wurde XSD verwendet. So lässt sich stark einschränken, wann ein Dokument valide ist und wann nicht. Dies wiederum vereinfacht das verfassen korrekter Regeln. Mit der JAXB-Bibliothek ist zudem eine gute und konsistente Bibliothek für den Umgang mit XML und XSD in Java verfügbar (siehe Abschnitt 7). Da diese ohnehin für die Verarbeitung der RMF-Daten genutzt wurde, liegt es nahe, sie auch an dieser Stelle einzusetzen.

Das Regelfile beschreibt zwei Dinge: *Requests,* die an die RMF gestellt werden und *Regeln,* die sich auf diese Requests beziehen. Zu einem Request gehört dabei ein eindeutiger Name, mit dem Regeln auf ihn verweisen können, sowie die RMF Ressource und die RMF ID. Außerdem ist ein Key nötig, falls der Request eine Liste von Werten zurückliefert. Abbildung 8.1 zeigt zwei Data Views in RMF PM: Links wird nur ein einzelner Wert zurückgegeben und angezeigt, rechts hingegen eine Liste. Wird eine solche Liste zurückgegeben, muss im Regel-File ein Key angegeben werden, der den gewünschten Wert in der Liste identifiziert (hier z.B. „*DA11*“, ein Datenbankname).

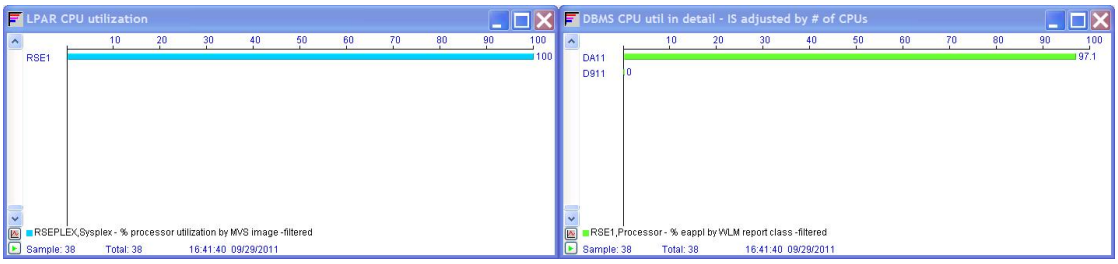


Abbildung 8.1.: RMF Daten als einzelner Wert (links) und als Liste (rechts)

Zu einem Regel-Tag gehört die Angabe von Klasse und Methode auf die sich die Regel beziehen soll. Zudem ist es nötig anzugeben, ob die Regel aktiv ist (`active`), wie oft sie geprüft werden soll (`check`), auf wie viel Prozent der betreffenden Methodenaufrufe sie angewendet werden soll (`cover`) und was ihre Empfehlung (`wait`) ist. Außerdem sind eine oder mehrere Bedingungen anzugeben, die *alle* zutreffen müssen, damit die Regel angewendet wird.

Diese Eigenschaften und Beziehungen lassen sich in XSD darstellen. So gewährleisten `key`-Contstrains, dass die Namen von Requests und Regeln eindeutig sind; die Verwendung von Datentypen (z.B. `double`) und Restriktionen auf sie (z.B. `maxInclusive` oder `minInclusive`) sorgen dafür, dass Attribute korrekt sind (Harold and Means, 2003, Seite 269 ff.) - z.B. darf das Attribut `cover` nur ein `double` Wert zwischen 0 und 1 annehmen.

Wird eine Regel angewendet, so gibt sie zuerst einmal nur eine *Empfehlung*. Das Lastkontrollsystem sammelt alle Empfehlungen der zutreffenden Regeln und wartet dann (je nach Einstellung im Properties-File) die maximale, die minimale, die aufsummierte oder durchschnittliche Zeit aller Empfehlungen.

8.3. Laden der Regeln

Damit die Regeln während der Ausführung verändert werden können, muss die Regel-Datei nachgeladen werden. Um dies zu realisieren, wurde die Apache Commons IO Bibliothek verwendet (näheres in Apache, 2010). Diese ermöglicht (u.a.), eine Datei auf Änderungen hin zu Überwachen. Wird die überwachte Datei editiert, so werden registrierte Listener-Objekte aufgerufen, die nun auf diese Änderungen reagieren können. Im Falle der Lastkontrolle werden dann die Regeln bzw. Requests neu geladen. So können z.B. Regeln während der Ausführung aktiviert und deaktiviert werden.

Das Nachladen der Regeln wirkt sich jedoch nicht auf die Instrumentierung aus. Da die Klassen nur einmalig beim Laden manipuliert werden können, sind die verfügbaren Regeln zu diesem Zeitpunkt entscheidend. Wird eine Regel für eine neue Klasse während der Laufzeit hinzugefügt, so hat dies keinen Effekt - die Anwendung muss hierfür zuerst neu gestartet werden. Lediglich Änderungen an den Parametern `wait`, `cover`, `active` und `check` von bereits *bestehenden* Regeln wirken sich während der Laufzeit aus.

Listing 8.1 zeigt einen Code-Ausschnitt, der die Überwachung eines Files startet. Hierfür sind nur wenige Zeilen Code nötig. Die Apache Bibliothek erledigt diese Aufgabe performant und fehlerfrei, weshalb diese Lösung vor dem entwickeln einer eigenen bevorzugt wurde.

Algorithmus 8.1 Überwachung von Dateien mit der Apache Commons IO Bibliothek

```
1  final File file = new File(Settings.get("rules"));
2
3  FileAlterationObserver observer =
4          new FileAlterationObserver(file.getParent(), new FileFilter() {
5              public boolean accept(File pathname) {
6                  return file.equals(pathname);
7              }
8          });
9
10 observer.addListener(new RuleManager());
11
12 FileAlterationMonitor monitor = new FileAlterationMonitor(5000);
13 monitor.addObserver(observer);
14 monitor.start();
```

8.4. Beispiel

Algorithmus 8.2 Auszug eines Regelwerks für das Lastkontrollsystem

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ruleSet
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:noNamespaceSchemaLocation="\Rules.xsd">
5      <request name="CPUForDB" resource="RSE1,*,PROCESSOR" id="8D27B0" key="DA11"/>
6      <rule
7          name="LIMITED_CPU"
8          class="java.sql.CallableStatement"
9          method="execute"
10         active="false"
11         check="45"
12         cover="0.35"
13         wait="1000" >
14         <condition request="CPUForDB" operator="&gt;" comparedTo="40"/>
15     </rule>
```

Listing 8.2 zeigt auszugsweise ein Regelwerk für das Lastkontrollsystem in XML und ist im Folgenden erklärt. Das Beispiel definiert einen Request zur RMF namens „`CPUForDB`“ sowie eine Regel namens „`LIMITED_CPU`“, die sich auf diesen Request bezieht.

- Zeile 1 bis 5 zeigen die XML-Deklaration, sowie die Einbindung des XSD-Files `Rules.xsd`. Dieses definiert die Syntax der Regeln.
- Zeile 5 zeigt die Definition eines RMF-Requests. Er kann im weiteren mit dem Schlüssel `CPUForDB` von Regeln referenziert werden. Das Lastkontrollsystem wird ihn in regelmäßigen Abständen ausführen und die Ergebnisse vorhalten.
- Zeile 6 bis 13 zeigen die Definition einer Regel, mit folgenden Eigenschaften:
  - Zeile 7 definiert ihren Namen (z.B. für Logging, aber auch als interner Schlüssel im Programm).
  - Zeile 8 und 9 zeigen die Klasse und deren Methode (ohne Parameter), auf die die Regel anwendet werden soll. Diese Methode wird zu Programmbeginn vom Agenten instrumentiert.
  - Zeile 10 zeigt, dass diese Regel derzeit ausgeschaltet ist, somit also nicht angewendet wird. Dies wirkt sich nicht auf die Instrumentierung, sondern lediglich auf die Ausführung der Regel aus.
  - Zeile 11 definiert, wie oft die Regel nacheinander geprüft werden soll, falls sie zuvor zutrifft. Die Zahl 45 bedeutet also, dass die Regel für einen einzigen Methodenaufruf bis zu 45-mal geprüft und angewendet wird. Danach

wird das Statement zugelassen. Dies soll Starvation von einzelnen Statements verhindern.

- ▷ Zeile 12 gibt an, auf wie viel Prozent der betreffenden Methodenaufrufe die Regel angewendet wird.
- ▷ Zeile 13 ist die Wartezeit in Millisekunden, die von der Regel empfohlen wird, falls sie zutrifft.

▷ Zeile 14 definiert eine Bedingung, die gelten muss, damit die Regel angewendet wird. In diesem Fall muss das Ergebnis des Requests `CPUForDB` größer als 40 sein, damit die Bedingung zutrifft. Es können mehrere solcher Bedingungen angegeben werden, die alle zutreffen müssen.

▷ Zeile 15 schließt das Regelwerk.

9. Programmablauf und Architektur

9.1. Architektur

Die entwickelte Lastkontrollschicht besteht aus drei Packages, welche die drei Hauptaspekte (Regelmechanismus, Instrumentierung und Sammeln von Performance-Daten) widerspiegeln. Das Package `com.ibm.jdbc.load.control` dient als Einstiegspunkt. Hier findet sich der Agent (samt `premain`-Methode), der die Instrumentierung und Bytecode-Manipulation vornimmt. Im Package `com.ibm.jdbc.load.control.rule` findet sich der Regelmechanismus. Dieses Package enthält Klassen, um Regeln zu laden und auszuwerten. In `com.ibm.jdbc.load.control.data` finden sich Klassen, die das Sammeln von Performance-Daten von der RMF übernehmen.

Abbildung 9.1 zeigt die wichtigsten Zusammenhänge der Lastkontrollschicht anhand eines gekürzten[1] UML-Class-Diagramms.

[1] In der Abbildung sind keine Eingabeparameter und Rückgabewerte dargestellt. Außerdem sind nur die wichtigsten Methoden abgebildet.

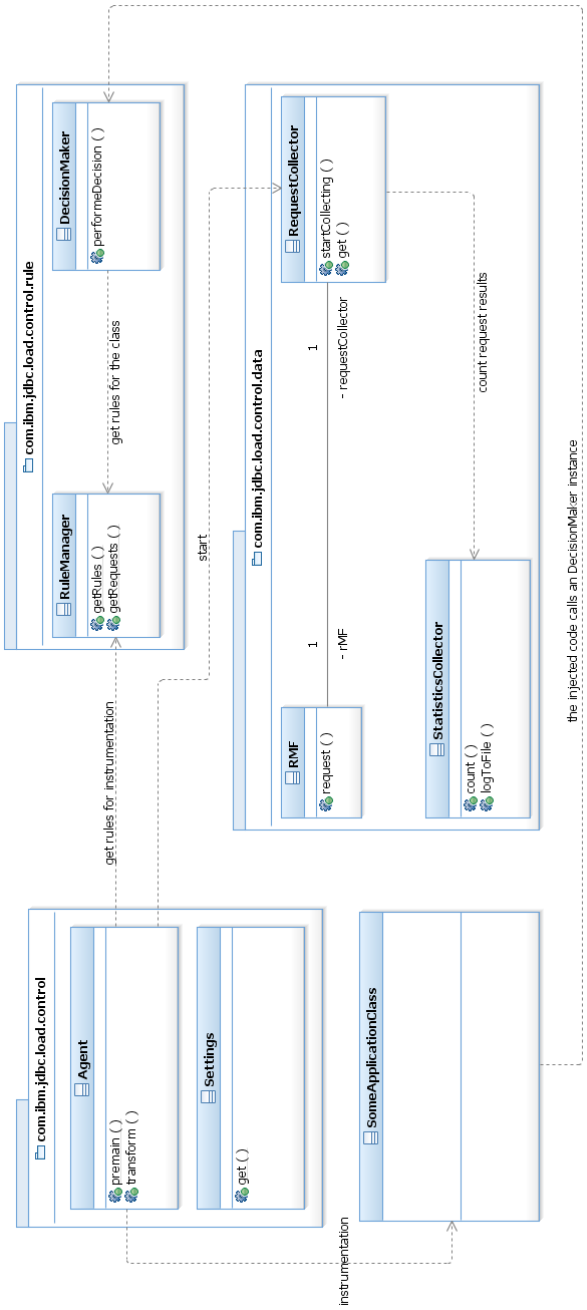


Abbildung 9.1.: Umriss der Architektur der Lastkontrollschicht

9.2. Programmablauf

Im Folgenden soll der Programmablauf der Applikation skizziert werden. Da die Anwendung teilweise mit parallelen Threads arbeitet, können sich verschiedene Schritte mehrfach wiederholen bzw. gleichzeitig ausgeführt werden. Abbildung 9.2 zeigt eine grafische Aufbereitung durch ein UML-Activity-Diagramm.

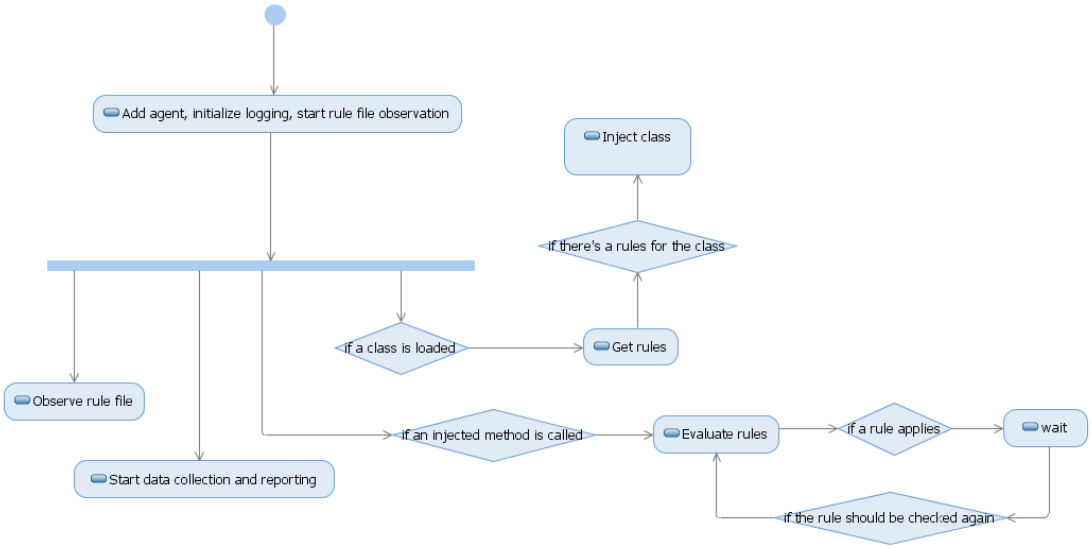


Abbildung 9.2.: Programmablauf (als UML Activity Diagramm)

1. Start der eigentlichen Applikation mit dem Lastkontrollsystem. Hinzufügen des Agenten für die Instrumentierung und initialisieren des Logging-Frameworks Log4J.

2. Start der Hintergrundthreads:

   a) Der *Collection*-Thread sammelt ab diesem Zeitpunkt Performance-Daten von der RMF. Hierzu synchronisiert er sich mit der RMF und fragt alle Requests ab, die in den Regeln definiert sind. Er führt jeden dieser Requests aus und hält dessen Ergebnis in einer Map eines Singletons vor. Der Key ist dabei der Name des Requests, wie er in den Regeln angegeben wurde. Außerdem hält der Collection-Thread eine Map vor, welche die Änderung zwischen dem letzten und dem aktuellen Wert des Requests enthält.
   Zudem sammelt und schreibt dieser Thread Logging-Informationen. Dies dient der Überprüfung des Programmablaufs und kann im Properties-File abge-

schaltet werden[2]. Es wird für jeden Request ein File geschrieben, in dem der aktuelle Wert für das Intervall festgehalten wird. So lassen sich aussagekräftige Diagramme zur Auswertung der Applikation erstellen[3].

b) Der *FileMonitor*-Thread überwacht mit Hilfe der Apache Commons IO Bibliothek das Regelfile auf Änderungen.

3. Wird eine Klasse geladen, so passiert diese den hinzugefügten Agenten. Er sucht daraufhin nach einer passenden Regel für die Klasse bzw. deren Methoden. Ist mindestens *eine* Regel für eine Methode dieser Klasse vorhanden, so wird diese instrumentiert. Ist keine Regel vorhanden, wird die Klasse nicht verändert.

4. Beim Aufruf einer instrumentierten Methode wird zuerst der eingefügte Programmcode zu Beginn der Methode ausgeführt. Er erzeugt eine Instanz des sogenannten *DecisionMaker* und ruft dessen Methode `performDecision` auf.

   a) Der DecisionMaker prüft, ob aktive Regeln für die Methode vorhanden sind. Wenn nicht, geschieht nichts weiter.

   b) Wenn aktive Regeln vorhanden sind, so werden diese ausgeführt und deren Empfehlungen (warten oder nicht und wenn ja, wie lange) gesammelt.

   c) Anhand der eingestellten Strategie wird entweder die minimale, die maximale oder die durchschnittliche Empfehlung ausgeführt - also gewartet.

   d) Soll die Regel mehrmals geprüft werden, werden die vorherigen zwei Schritte wiederholt.

[2] Nur das eigentliche - aber aufwändige - *Schreiben* der Datei kann abgestellt werden.
[3] Wie in Abschnitt 11.1 vorgestellt, kann dies auch automatisiert geschehen.

Teil IV.

Validierung & Abschließende Betrachtungen

10. Performance und Qualität

10.1. Performance

Lastkontrolle bedeutet zusätzlichen Aufwand. Die Instrumentierung von Klassen, das Abfragen von Performance-Daten und das Auswerten von Regeln - jeder dieser Schritte benötigt Ressourcen.

Im Folgenden sollen drei Fragen geklärt werden, die Aufschluss darüber geben, wie das Lastkontrollsystem die Performance beeinflusst: **Wie lange dauert ein Methodenaufruf eines Workloads...**

1. **...ohne die Lastkontrolle?**
2. **...mit der Lastkontrolle, aber nur mit deaktivierten Regeln?**
3. **...mit der Lastkontrolle und aktiven Regeln?**

Um diese Fragen zu klären, wurden zwei Tests durchgeführt. Zum Einen wurde ein spezieller Workload implementiert, der lediglich ein paar Sekunden pausiert und keine Datenbank nutzt. Dadurch ist die Ausführungszeit des Workloads konstant: Netzwerkverkehr und Datenbank spielen keine Rolle. Dieser Workload dokumentiert außerdem die Zeitspanne, die seine `execute`-Methode benötigt, in einem CSV-File[1]. Aus dieser Datei lassen sich dann die durch die Lastkontrolle verursachten Zeiteinbußen entnehmen. Zum Anderen wurde die Applikation mit dem Profiling-Tool VisualVM (näheres unter `http://visualvm.java.net/`) untersucht. Dies gibt u.a. Aufschluss über die lokale Auslastung der CPU und den CPU-Bedarf der einzelnen Methoden.

Tabelle 10.1 zeigt die Ausführungszeiten der `execute`-Methode eines Workloads mit 1000 Millisekunden Wartezeit. Es lässt sich erkennen, dass die Ausführungszeit der Methode um ca. 14 ms steigt, wenn die Lastkontrolle zugeschaltet wird, sie steigt um weitere 2 ms, wenn eine Regel aktiviert wird (ohne die Wartezeit der Regel die 1000 ms beträgt).

[1] Zwar schreibt auch das Lastkontrollsystem die Zeiten mit, die eine instrumentierte Methode zur Ausführung benötigt, jedoch ist als Vergleichswert insbesondere die Zeit *ohne* das Lastkontrollsystem interessant.

Im Falle mehrerer Regeln (hier im Test 10) ist die Performance der Lastkontrolle ebenfalls nahezu konstant, wie die letzte Spalte der Tabelle zeigt.

Ausführungszeit	Ohne LK	Mit LK (ohne Regeln)	Mit LK (eine Regel)	Mit LK (10 Regeln)
Durchschnitt (in ms)	1000	1016	1018 (+ Wartezeit: 2018)	1031 (+ Wartezeit: 2031)
Minimum (in ms)	1000	1000	1000 (+ Wartezeit: 2000)	1000 (+ Wartezeit: 2000)
Maximum (in ms)	1000	1047	1078 (+ Wartezeit: 2078)	1141 (+ Wartezeit: 2141)
Std. Abweichung	0	12.9	17.9	33.5

Tabelle 10.1.: Ausführungszeiten eines Workloads im Vergleich

Der zusätzliche Zeitbedarf durch die Lastkontrolle ist also gering. Das bloße Zuschalten benötigt im Schnitt 16 ms mehr. Dies ist insofern besonders wenig, als dass schon die Ausführung ohne Lastkontrolle, zeitweise eine maximale Ausführungszeit von 1015 ms hatte. Kommen in der Realität noch Datenbank und Netzwerkverkehr als (mehr oder weniger) zufällige Faktoren hinzu, so spielen 16 ms kaum eine Rolle. Mit einer eingeschalteten Regel (die *immer* 1000 ms wartet), ist die Zeit ebenfalls konstant. Subtrahiert man die 1000 ms Wartezeit der Regel, so erhält man nahezu die gleichen Zeiten wie ohne eingeschaltete Regeln.

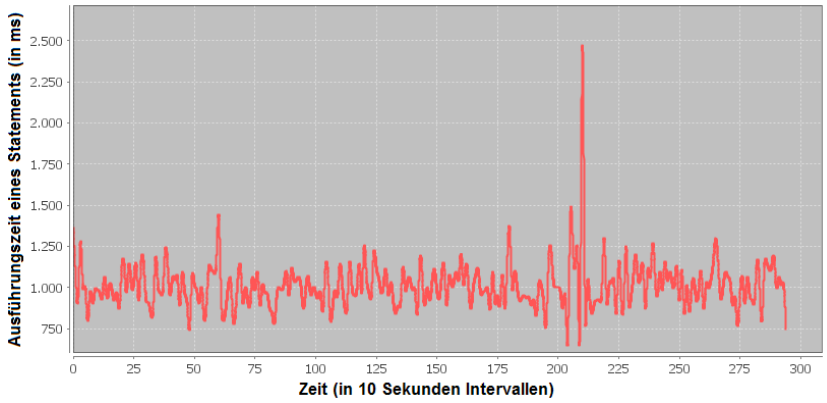


Abbildung 10.1.: Ausführungszeiten eines Join-Workloads ohne Lastkontrolle

Der Einfluss der Lastkontrolle auf die Performance der Anwendung ist also sehr gering. Abbildung 10.1 zeigt die Ausführungszeiten eines Workloads ohne Lastkontrolle auf einer Datenbank. Es ist zu erkennen, dass die Zeiten stark schwanken (zwischen 670 ms und 2500 ms). Bei Schwankungen der Größenordnung von mehreren hundert Millisekunden (bedingt durch Datenbank, Netzwerkverkehr und Client), können diese ~16 ms vernachlässigt werden.

Abbildung 10.2 zeigt den CPU-Bedarf der Applikation auf einem Client-Rechner[2] laut VisualVM[3],

1. ohne das Lastkontrollsystem (links)

2. und mit dem Lastkontrollsystem sowie mit einer eingeschalteten Regel (rechts).

Auf der Y-Achse ist die prozentuale Auslastung der CPU zu sehen, auf der X-Achse jeweils die Zeit[4]. Die Messungen wurden mit einem Workload ausgeführt, der einen Join auf einer Datenbank abfragt. Der Workload lastet die CPU des Systems dabei bis ca. 90% aus bzw. mit einer aktiven Regel bis ca. 50%. Es ist zu erkennen, dass der CPU-Bedarf mit der Lastkontrolle zwar anfänglich steigt, jedoch immer noch niedrig ist (zwischen 10% und 20%). Im weiteren Verlauf ist kaum ein Unterschied zwischen beiden Verläufen zu erkennen. Der CPU-Bedarf mit und ohne Lastkontrolle bewegt sich jeweils bei ca. 10%. Dieses Ergebnis zeigt somit ebenfalls, dass die Lastkontrolle nur wenig Zusatzlast bedeutet.

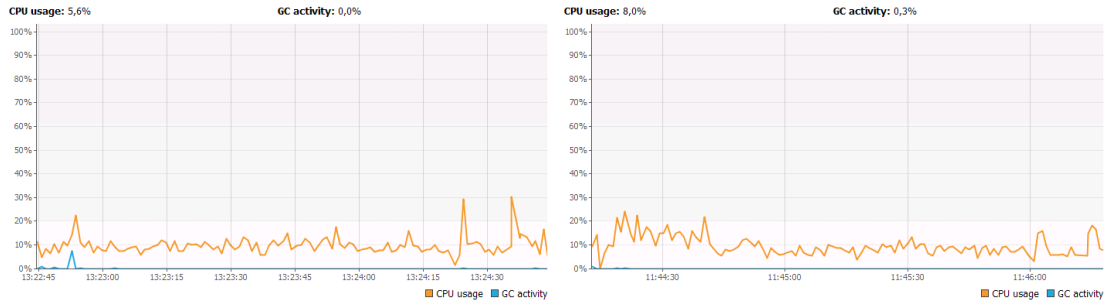


Abbildung 10.2.: Lokale Belastung der CPU mit und ohne Lastkontrolle

Tabelle 10.2 zeigt ausschnittsweise die prozentualen Anteile der einzelnen Methoden der Lastkontrolle an der CPU-Auslastung. Methoden des Workloads werden nicht dargestellt. Aus der Tabelle lässt sich erkennen, dass die Lastkontrolle nur einen sehr geringen Anteil an der gesamten CPU-Auslastung hat (unter 5%). Außerdem zeigt die Tabelle die rechenaufwändigen Methoden - hier könnte bei Bedarf eine Anpassung für Performance-Optimierung vorgenommen werden.

[2] Der Client-Rechner war in diesem Fall ein Lenovo T61p Laptop, mit 2,4 GHz Duo Core.

[3] Leider lassen sich die gemessenen Werte nicht aus VisualVM exportieren. Daher wurde für die Darstellung der Messungen auf Screenshots zurückgegriffen. Aus diesem Grunde unterscheiden sich diese beiden Diagramme von ihrer Optik her von den übrigen in dieser Arbeit.

[4] Beide Messungen wurden zu verschiedenen Zeiten ausgeführt, daher unterschieden sich die Uhrzeiten, die in der X-Achse abgebildet sind. Beide Ausschnitte sind jedoch vergleichbar lang, jeweils ca. 2 Minuten.

Methode	CPU-Auslastung	Bemerkung
`org.apache...FileAlterationMonitor.run()`	4.8%	Überwacht das Regel-File auf Änderungen.
`com.ibm...RMF.request()`	0.1%	Fragt die RMF ab.
`com.ibm...Agent.injectMethod()`	< 0.1%	Führt Byte-Code-Manipulation aus.
`com.ibm...StatisticsCollector.logToFile()`	< 0.1%	Schreibt Statistiken in eine Datei.

Tabelle 10.2.: Prozentuale Anteile der Lastkontrollmethoden an der CPU-Auslastung

10.2. Qualität

Das Lastkontrollsystem ist lediglich als ein erster Entwurf gedacht - ein sogenanntes *Proof of Concept*. Trotzdem sollte eine gewisse Qualität garantiert werden. Dies macht es leichter, die Software in einem späteren Projekt zu verwenden, besonders für dritte - sie könne sich auf die Qualität der bestehenden Implementierung verlassen. Außerdem erleichtert eine gute Qualität auch die Validierung der Applikation (die im folgenden Kapitel ausführlich dargestellt ist). So werden schon vorab viele Fragen geklärt: Geben die Methoden die erwarteten Werte zurück? Werden unerwartete Exceptions geworfen? Werden alle Klassen richtig initialisiert?

Um die Qualität der Lastkontrollschicht sicher zu stellen, wurden Unit-Tests für die entwickelten Klassen geschrieben. Abbildung 10.3 zeigt ein Screenshot mit der Darstellung der Code-Coverage durch die Unit-Tests mit dem Eclipse Tool EclEmma[5].

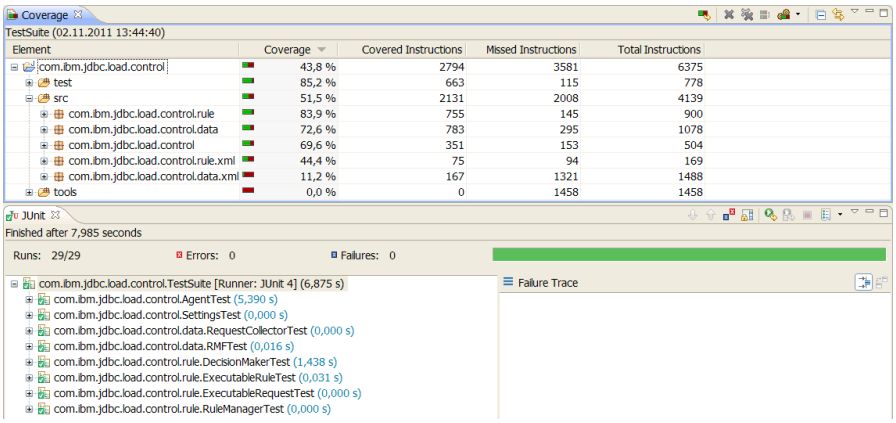


Abbildung 10.3.: Code-Coverage der JUnit-Tests für die Lastkontrollschicht

[5] siehe `http://www.eclemma.org/`

Wie in Abbildung 10.3 ersichtlich, sind die Tests erfolgreich und decken ca. 44% des gesamten Codes ab. Die relevante Code-Abdeckung liegt jedoch deutlich höher. Denn weder die Abdeckung der Tests selbst (wobei diese mit 94% hoch ist), noch die Abdeckung der Tools sind interessant. Die Tools dienen lediglich der Ausführung von Tests mittels ANT-Skripten, was im folgenden Kapitel näher beschrieben ist. Für die eigentliche Anwendung spielen sie keine Rolle. Uninteressant ist ebenfalls die Abdeckung der Packages `com.ibm.jdbc.load.control.rule.xml` und `com.ibm.jdbc.load.control.data.xml` - diese Pakete enthalten ausschließlich von JAXB generierte Klassen.

Die relevanten Packages, die Code für die Anwendung enthalten sind also

- ▷ `com.ibm.jdbc.load.control`[6] welches zu 70% abgedeckt wird,
- ▷ `com.ibm.jdbc.load.control.rule`[7] welches zu 84% abgedeckt wird und
- ▷ `com.ibm.jdbc.load.control.data`[8] welches zu 73% abgedeckt wird.

Die Codeabdeckung für diese Packages ist also ausreichend hoch.

[6] Hier ist der Agent und die Klasse für die Settings enthalten.
[7] Hier sind Klassen zum Auswerten und Durchführen der Regeln enthalten.
[8] Hier sind Klassen für das Sammeln von Performance-Daten enthalten.

11. Validierung

Dieses Kapitel beschreibt die Validierung der Applikation. Für *gute* Softwareentwicklung sind ausführliche Tests essentiell - was nicht zuletzt in einem Testteam wie dem des Performance Experts deutlich wird. Die folgenden Abschnitte zeigen daher ausführlich die Vorgehensweise und Ergebnisse dieser Tests. Dies soll sie - auch für dritte - nachvollziehbar und verlässlich machen. Es werden Fragen geklärt wie: Funktioniert die Lastkontrolle überhaupt? Wie gut sind die erzielten Ergebnisse? Wie wirken sich die einzelnen Parameter der Regeln aus?

11.1. Vorgehensweise & Automatisierung der Tests

Versuchsaufbau, Beobachtung und Deutung - das waren während der Schulzeit stets die Schritte, denen ein Versuch im Chemieunterricht folgte. Der *Versuchsaufbau* (eventuell mit Skizze) sollte das Experiment dokumentieren und wiederholbar machen. Die *Beobachtung* sollte für spätere Studien festhalten, was damals geschah und die *Deutung* sollte widerspiegeln, wie die Ergebnisse damals verstanden wurden.

Tests in der Softwaretechnik folgen ähnlichen Kriterien. Sie müssen wiederholbar sein, um bei jeder Änderung am Programm erneut durchgeführt werden zu können; ihre Ergebnisse müssen gespeichert werden, um beispielsweise die Entwicklung von Defects zu verfolgen; sie müssen gedeutet werden, um zu wissen ob, sie erfolgreich/ausreichend waren oder nicht. Die Tests der Lastkontrollschicht wurden daher möglichst weitgehend automatisiert. Dies macht sie wiederholbar und wohl definiert.

Der *Versuchsaufbau* wurde durch ANT-Skripte definiert. Mit ihnen wurde der Testworkload gestartet, erwartete Werte kontrolliert und gesteuert, wann welche Regel aktiviert oder verändert werden soll. So kann ein beliebiger Testfall erstellt werden.

Die *Beobachtung* wurde ebenfalls über dieses ANT-Skript gesteuert. Eine eigens entwickelte Java-Klasse berechnet zu festgelegten Intervallen Schlüsselwerte wie Minimum, Durchschnitt oder die Standardabweichung der einzelnen Messwerte zueinander. Diese Werte werden in eine Datei geschrieben und am Ende des Tests automatisch mit den Logs der Lastkontrollschicht in einem Ordner archiviert. Mittels der BIRT-Bibliothek

(siehe `http://eclipse.org/birt/phoenix/`) werden außerdem automatisch Diagramme aus den aufgezeichneten CSV-Dateien generiert und mit archiviert. Lediglich die *Deutung* der Ergebnisse bleibt dem Tester überlassen. Jedoch wird diese maßgeblich dadurch erleichtert, dass sowohl Diagramme als auch mathematische Kennzahlen automatisch bereitgestellt werden.

Abbildung 11.1 zeigt einen Ausschnitt eines solchen Testfalles in ANT. Zeile 2 baut zunächst die Applikation. So wird ein neuer und sauberer Stand garantiert. In Zeile 5 wird vor Beginn des Tests die Ausgangslast kontrolliert, die hier unter 3% betragen soll. Zeile 12 startet den Workload, mit dem die Lastkontrolle getestet werden soll. Der Tag in Zeile 13 pausiert kurz und verschafft dem Workload so Zeit, um zu starten. Der Aufruf des Targets `watch` in Zeile 14 startet die Java-Klasse, welche für den angegebenen Zeitraum (im Ausschnitt nicht zusehen) Minimum, Maximum, Durchschnitt und Standardabweichung der Messwerte berechnet. Zeile 15 bis 19 aktivieren eine Regel. Daraufhin wird in Zeile 20 wieder gewartet, damit die Änderungen wirken können. In Zeile 21 werden für die vorgegebene Zeitspanne erneut die mathematischen Schlüsselwerte berechnet (`watch`). Abschließend werden in Zeile 22 sämtliche Logs, CSV-Dateien und Regeln in einen neuen Ordner kopiert. In diesem Schritt werden auch die Diagramme zu den CSV-Dateien generiert.

Algorithmus 11.1 Ausschnitt eines Testfalles in ANT

```
1  <target name="test">
2      <ant antfile="./scripts/build.xml" dir="." />
3      <copy file="${test.dir}/TestTemplate.xml"
4            todir="${staf.path}/services/db2wkl/${load.control.settings}"/>
5      <ant antfile="scripts/checkCondition.xml">
6          <property name="check.resource" value="RSE1,*,PROCESSOR"/>
7          <property name="check.id" value="8D27B0"/>
8          <property name="check.operator" value="&lt;"/>
9          <property name="check.value" value="3"/>
10         <property name="check.key" value="DA11"/>
11     </ant>
12     <ant antfile="scripts/startWorkload.xml" />
13     <ant antfile="scripts/sleep.xml" />
14     <ant antfile="scripts/watch.xml" />
15     <ant antfile="scripts/changeSetting.xml">
16         <property name="change.rule" value="Limited_ZIIP"/>
17         <property name="change.attribute" value="active"/>
18         <property name="change.value" value="true"/>
19     </ant>
20     <ant antfil="scripts/sleep.xml" />
21     <ant antfile="scripts/watch.xml" />
22     <ant antfile="./scripts/createCharts.xml" dir="." />
23 </target>
```

So verfasste Tests können leicht wiederholt werden und auch ohne die Aufmerksamkeit des Entwicklers im Hintergrund laufen. Den einzigen Punkt, den diese Tests nicht kontrollieren können, sind andere Applikationen, die gleichzeitig auf der Datenbank operieren. Sie können lediglich erwartete Vorgaben testen und abbrechen, falls diese nicht zutreffen (siehe Zeile 5 des ANT-Skripts).

11.2. Testläufe

11.2.1. Generelle Funktionalität

Funktioniert das Lastkontrollsystem? - Dies ist die grundlegendste Frage. Abbildung 11.1 zeigt die Ergebnisse eines Testlaufs mit einem Stored Procedure Workload[1] und dem Lastkontrollsystem. Auf der X-Achse ist die Zeit in 10-Sekunden Intervallen abgetragen[2], auf der Y-Achse die prozentuale Auslastung der CPU durch die genutzte Datenbank.

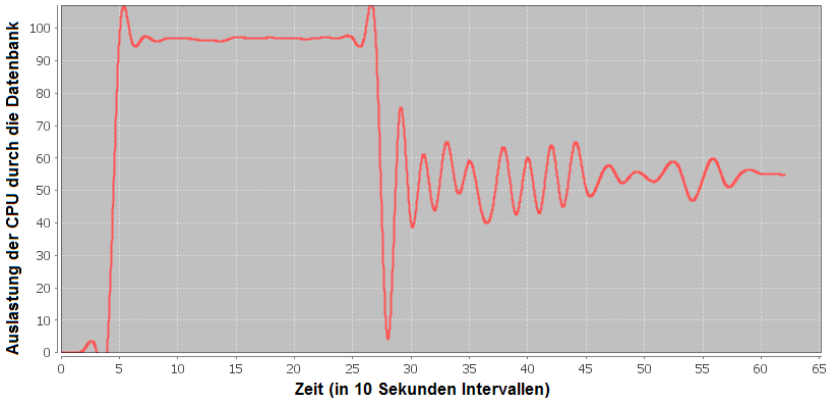


Abbildung 11.1.: Anteilige CPU-Auslastung der genutzten Datenbank

Man kann ab Intervall 27 einen deutlichen Einbruch der Kurve erkennen - die Auslastung der CPU geht drastisch zurück. Zu diesem Zeitpunkt wurde das Lastkontrollsystem eingeschaltet. In den folgenden Sekunden (Intervalle 30 bis 60) pendelt sich die Auslastung erkennbar um den Wert 54 ein. Jedoch ist auch eine starke Schwankung ($\sigma = 7,6$[3]) im Gegensatz zu den vorherigen Werten ohne die Lastkontrolle ($\sigma = 0,3$) zu erkennen[4].

Für den Testlauf wurde die in Abbildung 11.2 gelistete Regel verwendet. Anzumerken ist, dass obwohl das CPU-Limit auf 30% eingestellt ist, die tatsächlich Auslastung im Mittel bei 54% lag. Dies kann von mehreren Parametern abhängen. So wird die Regel z.B. nur auf jeden zweiten Methodenaufruf von `execute` angewandt (`cover` = 0,5). Würde dieser Wert höher sein, so wäre auch die Schwankung der Kurve größer. Daher muss ein Kompromiss zwischen den Einstellungen eingegangen werden (siehe hierzu Abschnitt 11.2.2).

[1] Der Workload ruft in mehreren Threads eine Stored Procedure auf, die einen Join ausführt. Dabei steigt die CPU-Auslastung stark an, wodurch sich dieser Workload besonders für den Test eignet.
[2] Dies ist die Zeitspanne, in der die RMF ihre Messdaten aktualisierte.
[3] Mit σ ist die Standardabweichung der einzelnen Messwerte zueinander gemeint.
[4] Diese Werte lassen sich dem mitgeschriebenen Log-Files entnehmen.

Algorithmus 11.2 Regel für den Testlauf

```
<request key="DA11" id="8D27B0" resource="RSE1,*,PROCESSOR" name="CPUForDB"/>
<rule
      wait="800"
      cover="0.5"
      check="40"
      active="true"
      method="execute"
      class="java.sql.CallableStatement"
      name="Limited_CPU_USE_FOR_DBMS_DISTRIBUTED">
      <condition comparedTo="30.0" operator="&gt;" request="CPUForDB"/>
</rule>
```

Der Zusammenhang zwischen CPU-Auslastung und dem Durchsatz lässt sich anhand von Abbildung 11.2 erkennen. Wie in der vorherigen Abbildung, bricht auch hier die Kurve um das Intervall 25 ein. Die Kurve sinkt dabei etwas früher als die der CPU-Auslastung. Dies hängt damit zusammen, dass zuerst die Anzahl der Statements durch Ausbremsen des Lastkontrollsystems verringert wird und in Folge dessen *dann* die CPU-Auslastung sinkt. Daher geht diese Kurve der anderen voraus.

Die Abbildung zeigt deutlich, dass die Auslastung der CPU direkt mit den durchgeführten Statements pro Zeiteinheit zusammenhängt. Je mehr Statements pro Zeiteinheit ausgeführt werden, desto höher ist die Last für das System. Umgekehrt bedeutet dies auch, dass das Lastkontrollsystem den Durchsatz (und damit die Applikation) klar ausbremst und verlangsamt.

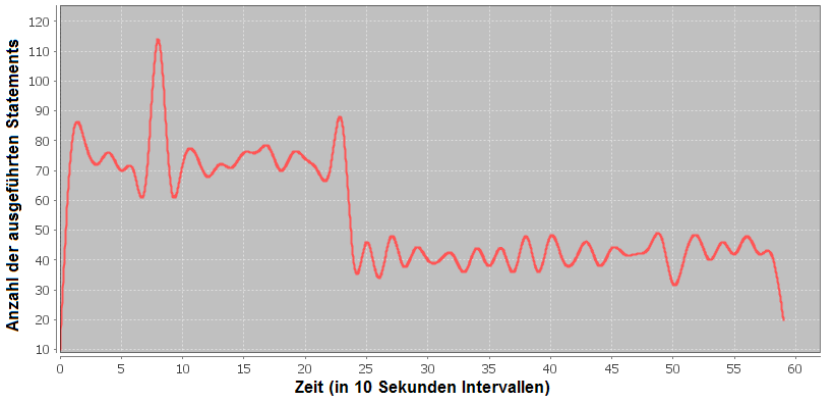


Abbildung 11.2.: Anzahl der Statements pro Intervall

Interessant ist auch der Zusammenhang in Abbildung 11.3 - wenn auch nicht so deutlich wie der Zwischen CPU-Auslastung und Durchsatz. Die Abbildung zeigt die durchschnittliche Ausführungszeit eines Statements (Y-Achse) in 10-Sekunden-Intervallen (X-Achse). Die Abbildung visualisiert also den Verlauf der Antwortzeiten der Datenbank.

Es ist zu sehen, dass die Ausführungszeit vor dem Intervall 25 nahezu gleichbleibend bei 11 Sekunden lag. Ab dem Einschalten der Lastkontrolle beginnt die Ausführungszeit zwar zu schwanken, sinkt aber im Mittel erkennbar (auf ca. 10 Sekunden pro Statement). Dies zeigt folgendes: Ist die Auslastung des Systems hoch (hier bei ca. 97% CPU-Auslastung), so ist auch die Antwortzeit höher (ca. 11 Sekunden) als bei einer niedrigeren Auslastung (ca. 10 Sekunden bei durchschnittlich 54% CPU-Auslastung).

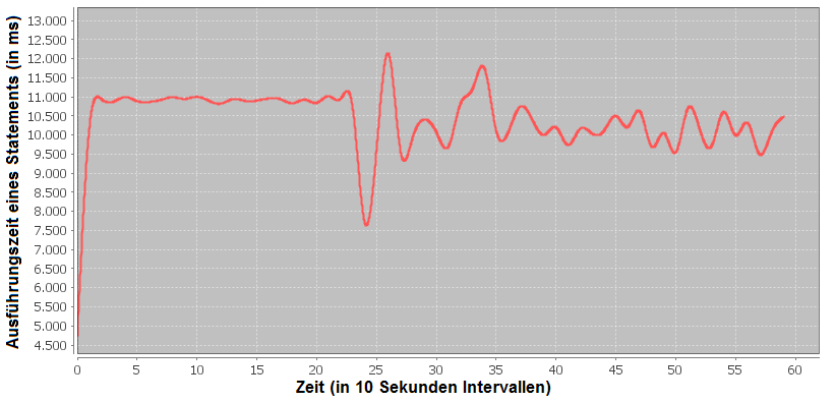


Abbildung 11.3.: Durchschnittliche Ausführungszeit pro Statement

All diese Ergebnisse verdeutlichen folgendes: Das Lastkontrollsystem funktioniert grundsätzlich. Es ist in der Lage, z.B. die Auslastung der CPU klar nach oben hin zu beschränken. Durch das Ausbremsen der Applikation wird jedoch auch der Durchsatz verringert - die Anwendung wird langsamer.

11.2.2. Güte der Ergebnisse

Ein Kriterium für die Güte der Lastkontrolle bzw. deren Ergebnisse, ist die möglichst geringe Abweichung der einzelnen Messwerte zueinander - die Kurve soll also nicht oszillieren. Wie aus Abbildung 11.1 des vorherigen Abschnitts zu entnehmen, schwankt die Auslastung nach dem Einschalten der Lastkontrolle merklich. **Womit aber hängt dies zusammen und wie kann dem entgegen gewirkt werden?**

Die Lastkontrolle ist *träge*: sie kann Last nicht voraussagen, sondern lediglich auf sie reagieren. Dadurch wird eine Applikation erst dann ausgebremst, wenn die Leistungsdaten die durch die Regeln festgelegten Werte überschritten haben. Ab dann wird solange „gebremst“, bis die Leistungsdaten wieder unterhalb der Grenzwerte sind. Hier greifen die Regeln erneut - der Kreislauf beginnt von neuem.

Die Werte, die von der Lastkontrolle beeinflusst werden, schwanken also zwangsweise *immer* - mehr oder weniger. Der Grad der Schwankung hängt maßgeblich von den eingestellten Regelparametern ab. Abbildung 11.4 zeigt einen halbstündigen Testlauf mit drei verschiedenen Regeleinstellungen.

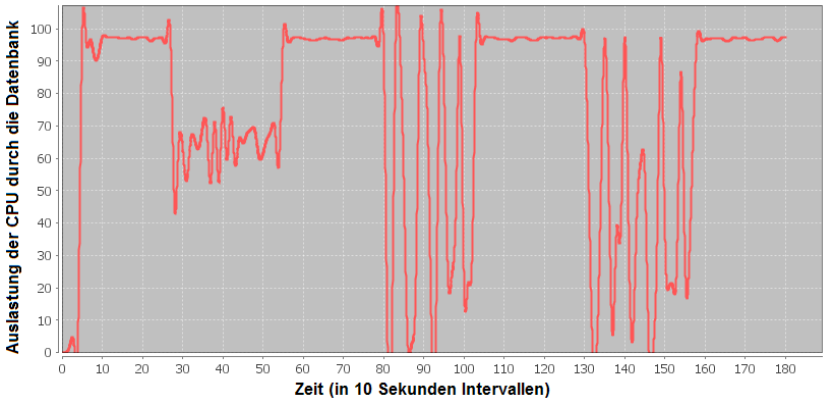


Abbildung 11.4.: Halbstündiger Test mit verschiedenen Regeleinstellungen

Die Abbildung lässt deutlich drei Bereiche erkennen:

1. **Intervall 30 bis 55:** Hier wurde eine „gute“ Regel (`cover` = 0.4, `wait` = 1050, `check` = 35) angewendet, weshalb die Schwankung relativ gering ist. Sie wurde als Vorlage für die anderen beiden Abschnitte genutzt und jeweils in einem Parameter leicht verändert.

2. **Intervall 80 bis 105:** Hier wurde dieselbe Regel mit einem höheren `cover`-Wert genutzt (zuvor 0.4, jetzt 0.8). Dadurch wurden doppelt so viele Statements reguliert, was heftige Ausschläge der Kurve zur Folge hat. Diese sinkt deutlich, da viele Statements ausgebremst werden, steigt aber auch ebenso deutlich an, da gleichzeitig viele Statements beim unterschreiten des Grenzwertes wieder freigegeben werden.

3. **Intervall 130 bis 155:** Hier wurde der `wait`-Parameter der Regel aus Intervall 30 bis 55 geändert: von zuvor 1050 Millisekunden auf 2000 Millisekunden. Auch hierdurch entstehen heftige Sprünge im Kurvenverlauf. Die Statements werden für einen sehr langen Zeitraum gebremst, weshalb die Last merklich einbricht. Ist die Wartezeit vorüber, werden alle gebremsten Statements ausgeführt, was wiederum zu einem sprunghaften Anstieg führt.

Welche Regelparameter beeinflussen also den Verlauf der Leistungswerte? - Entscheidend sind die Werte `check`, `cover` und `wait`. Abbildung 11.5 zeigt einen Testlauf mit drei Phasen, in denen je unterschiedliche - aber vergleichbar „gute“ - Parameterwerte genutzt wurden. Die Einstellungen waren wie folgt:

1. **Intervall 30 bis 55:** `cover = 0.4`; `check = 35`; `wait = 1050`;
2. **Intervall 80 bis 105:** `cover = 0.35`; `check = 35`; `wait = 1400`;
3. **Intervall 130 bis 155:** `cover = 0.4`; `check = 35`; `wait = 700`;

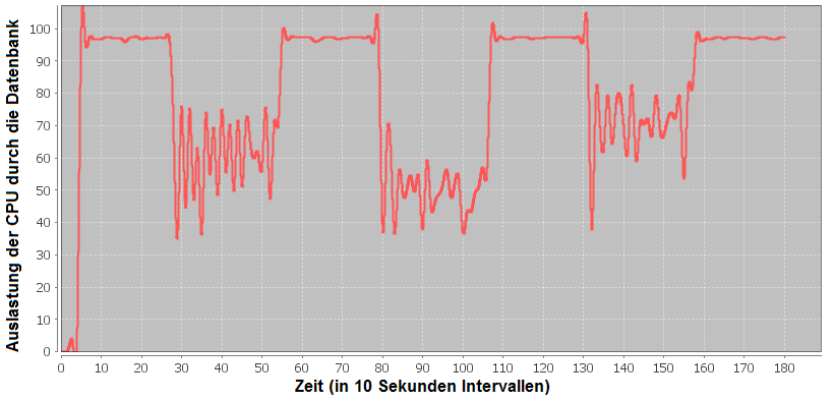


Abbildung 11.5.: CPU-Auslastung mit drei ähnlich *guten* Parametereinstellungen

Aus beiden Abbildungen lässt sich erkennen, wie die Parameter gemeinsam den Verlauf der Kurve beeinflussen. Während der Arbeit am Lastkontrollsystem ergeben sich folgende Erfahrungswerte, die zu guten Ergebnissen[5] geführt haben:

- ▷ Der Parameter `wait` sollte relativ gering (etwa 500 bis 1500 Millisekunden) eingestellt sein. Damit wird garantiert, dass Statements nicht zu lange gebremst werden, sondern weiter laufen können sobald sich die Lastentwicklung entspannt hat. So entstehen keine Einbrüche in der Kurve. Ist dieser Parameter eher hoch (ca. 1500 ms), so wird die Auslastung stärker gebremst als bei einem niedrigerem Parameter (vgl. Intervall 30 - 55 und Intervall 130 - 155).
- ▷ Der Parameter `cover` sollte bei ca. 0.5 eingestellt sein. Dies hat zur Folge, dass nur jedes zweite Statement von der Regel betroffen ist. So werden nie *alle* Statements gebremst, wodurch immer eine gewisse *Grundlast* herrscht.

[5] Also einem möglichst flachen Kurvenverlauf.

▷ Der Parameter `check` sollte so eingestellt sein, dass er zusammen mit der eingestellten `wait`-Zeit etwa einem Aktualisierungsintervall entspricht. So wird ein einzelnes Statement nie lange ausgebremst, sondern die Regel auf immer neue Statements angewendet.

Die Güte der Ergebnisse hängt jedoch nicht nur vom Lastkontrollsystem und seinen Parametern ab - auch die jeweilige *Anwendung* ist entscheidend. Es sind nur Anwendungen geeignet, die fortlaufend eine große Zahl an Datenbankstatements absetzen. Dies hat zwei Gründe:

1. Die Lastkontrolle ist träge, d.h. sie reagiert erst, wenn Last bereits entstanden ist. Setzt das Programm nur „ab und zu“ ein paar Statements ab, so erzeugen diese zwar Last, sind aber schon beendet, wenn das Lastkontrollsystem reagieren kann - es kommt zu spät.

2. Das Lastkontrollsystem kann nur eingreifen, wenn eine gewisse Anzahl an Statements abgegeben werden. Setzt die Applikation nur wenige, aber sehr aufwendige Anfragen ab (z.B. sehr große Joins über mehrere Tabellen), so lasten diese die Datenbank unweigerlich aus. Das Lastkontrollsystem kann sie zwar verzögern, aber ihre Last nicht verringern.

Abbildung 11.6 zeigt den Zusammenhang zwischen der Güte der Ergebnisse und der Anzahl an abgesetzten Statements. In der ersten Hälfte des Testlaufes wurde ein Workload verwendet, der eine größere Zahl von einfachen Join-Abfragen aufgibt (ohne Lastkontrolle ca. 500, mit Lastkontrolle ca. 300 in einem Intervall). In der zweiten Hälfte wurde ein Workload verwendet, der wenige, aber dafür aufwendigere Join-Abfragen durchführt (etwa 47 in einem Intervall, sowohl mit als auch ohne Lastkontrolle). Die verwendete Regel war in beiden Fällen dieselbe.

Es ist zu erkennen, dass die Lastkontrolle in der ersten Hälfte wesentlich besser gearbeitet hat als in der zweiten Hälfte (dort ist fast keine Veränderung zu erkennen). Während im Intervall 30 bis 50 die Auslastung deutlich sank (auf im Schnitt 36,5% bei $\sigma = 3,9$), ging die Auslastung im Intervall 100 bis 120 kaum merklich nach unten (von 97,9% bei $\sigma = 0,2$ auf 97,4% bei $\sigma = 0,5$). Dies liegt daran, dass das Lastkontrollsystem beim ersten Workload und seinen *vielen* Statements regulierend eingreifen konnte. Bei den langlaufenden Joins im zweiten Workload, war die Lastkontrolle hingegen machtlos - jeder für sich lastete das System schon aus.

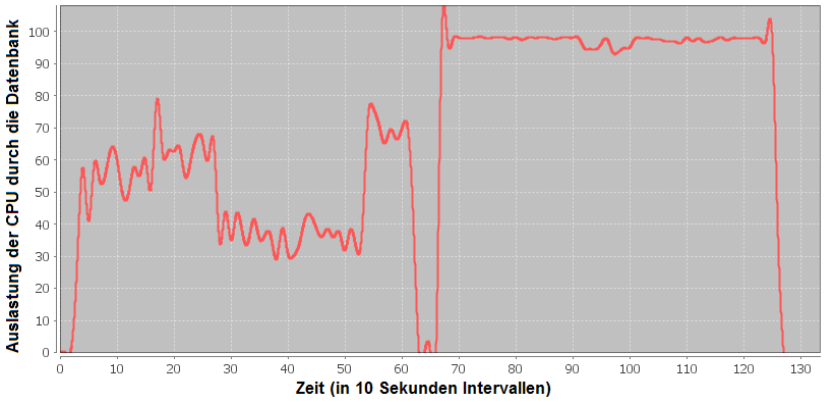


Abbildung 11.6.: Vergleich zweier Workloads mit vielen bzw. wenigen Transaktionen

11.2.3. Mehrere Rechner

Wie sich die Lastkontrollschicht auf einem einzelnen Rechner mit einer exklusiv genutzten Datenbank verhält, wurde im vorherigen Abschnitt dargelegt. Wie aber sind die Ergebnisse, wenn mit mehreren Rechner auf *einer* Datenbank gearbeitet wird?

Im Folgenden werden Tests mit zwei Rechnern beschrieben. Dabei wurde einmal das Szenario betrachtet, dass die Lastkontrolle auf beiden Rechnern gleichermaßen eingerichtet ist (siehe Abschnitt 11.2.3.1) und einmal das Szenario, dass nur einer der beiden Rechner die Lastkontrolle besitzt (siehe Abschnitt 11.2.3.2).

Der Test war dabei jeweils derselbe: Auf beiden Maschinen wurde quasi zeitgleich ein Workload gestartet, der mit einer Stored Procedure einen langen Join auf einer Datenbank ausführt. Anfänglich war die Lastkontrolle (sofern vorhanden) ausgeschaltet und wurde erst später (nach ca. 3 Minuten) zugeschaltet. So ist ihr Einfluss deutlich sichtbar.

11.2.3.1. Beidseitige Lastkontrolle auf zwei Rechnern

Abbildung 11.7 zeigt die CPU-Auslastung (Y-Achse) eines ca. 10-minütigen Testlaufs (X-Achse). Der Testlauf wurde mit zwei Rechnern ausgeführt, auf denen die Lastkontrolle gleichermaßen eingerichtet war. Im Intervall 25 bis 45 ist hier ein deutlicher Rückgang der CPU-Auslastung zu beobachten. Zudem oszilliert die Kurve sehr wenig ($\sigma = 4,0$). Die Auslastung liegt in diesem Intervall bei durchschnittlich 44%, was mit den Regeln gut zusammenpasst - hier waren 30% als Limit für die CPU-Auslastung vorgegeben.

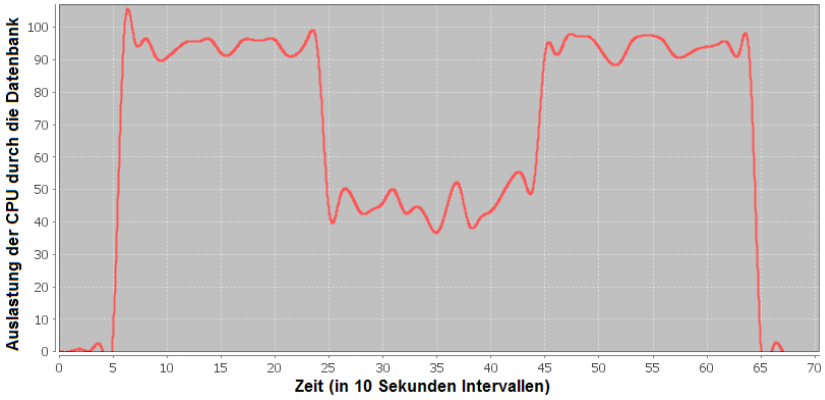


Abbildung 11.7.: CPU-Auslastung durch 2 Rechner mit beidseitiger Lastkontrolle

Fazit: Ist die Lastkontrolle auf mehreren Rechnern gleichermaßen eingerichtet, funktioniert sie auch in diesem Szenario. Beide Applikationen drosseln ihre Geschwindigkeit in annähernd gleichem Maße. Tabelle 11.1 zeigt die durchschnittliche Anzahl der ausgeführten Statements der beiden Maschinen in den drei Intervallen. Es ist zu erkennen, dass keine Maschine „zu kurz“ kommt, also deutlich weniger Statements durchführen kann als die andere. Die Lastkontrolle funktioniert folglich *fair* und erzielt gute Ergebnisse.

Maschine	**Intervall 5 bis 25**	**Intervall 25 bis 45**	**Intervall 45 bis 60**
local	1020	500	1010
remote	1100	560	1120

Tabelle 11.1.: Durchschnittliche Statementzahl mit beidseitiger Lastkontrolle

11.2.3.2. Einseitige Lastkontrolle auf nur einem Rechner

Abbildung 11.8 zeigt analog zu Abbildung 11.7 was passiert, wenn die Lastkontrolle nur einseitig auf *einem* der beiden Rechner installiert bzw. angeschaltet ist[6]. Es passiert *nichts*. Die Auslastung der CPU verändert sich über den gesamten Verlauf nicht, auch wenn die Lastkontrolle wie zuvor im Intervall 25 bis 45 eingeschaltet wird.

[6] In diesem Fall war die Lastkontrolle zwar auf beiden Rechnern installiert, auf dem zweiten jedoch ausgeschaltet. So erhält man jedoch für beide Maschinen, die von der Lastkontrolle geschriebenen Statistiken zur Auswertung.

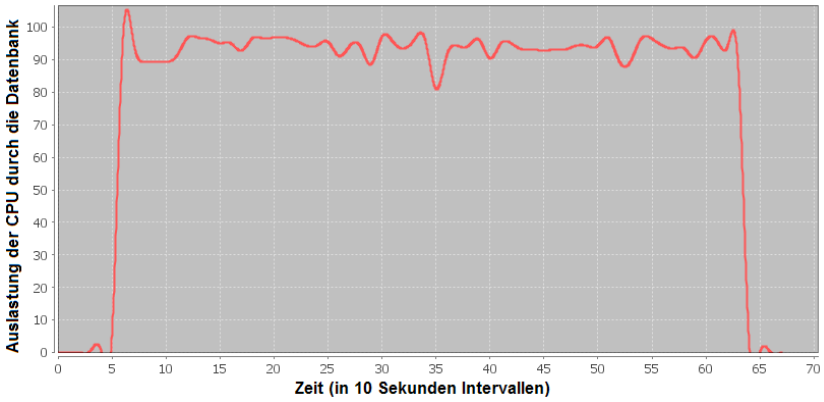


Abbildung 11.8.: CPU-Auslastung durch 2 Rechner mit einseitiger Lastkontrolle

Interessant ist in diesem Szenario die Anzahl der ausgeführten Statements. Tabelle 11.2 zeigt diese für die drei Intervalle. Es ist zu sehen, dass die Maschine mit der Lastkontrolle zwar weniger Statements ausführt, die Maschine ohne die Lastkontrolle hingegen mehr - beides hält sich die Waage. Daher ist hier kein Rückgang der Auslastung zu erreichen.

Maschine	**Intervall 5 bis 25**	**Intervall 25 bis 45**	**Intervall 45 bis 60**
mit Lastkontrolle	1095	470	1090
ohne Lastkontrolle	1035	1650	1030

Tabelle 11.2.: Durchschnittliche Statementzahl mit einseitiger Lastkontrolle

Die Applikation mit Lastkontrolle bremst deshalb nicht völlig ab, da die Regel nur auf 40% der Statements angewendet (`cover` = 0.4) wird, und das auch nicht für immer (`check` = 35). Daher hat auch diese Applikation eine Grundlast. Dies verhindert z.B., dass sie völlig „verhungert“ und gar keine Statements mehr absetzen kann.

12. Grenzen & Kritik

Das Lastkontrollsystem hat in der implementierten Form einige Nachteile bzw. Kritikpunkte. Diese wurden in den vorigen Abschnitten teilweise schon angesprochen und erklärt. Im Folgenden sollen sie kurz zusammengefasst werden.

- ▷ Das Lastkontrollsystem ist nur für Anwendungen mit häufigen und kurzen Statements geeignet. Lasten bereits einzelne Statements (z.B. aufwändige Join-Abfragen) das System aus, so kann die Lastkontrolle nicht eingreifen. Sie kann die Statements nur herauszögern, nicht aber ihren eigentlichen Aufwand verringern (siehe Ende Abschnitt 11.2.2).

- ▷ Die Implementierung des Rule-Files ist bislang einfach gehalten. Sie lässt z.B. keine Vererbung oder komplex verknüpfte logische Bedingungen zu. Für kleine Regel-Dateien bzw. den hier vorgestellten Prototyp ist dies ausreichend. Jedoch könnten bei (sehr) langen Regel-Dateien solche Funktionen wünschenswert sein.

- ▷ Die Güte der Ergebnisse hängt von den Einstellungen und den jeweils verwendeten Workloads ab. D.h., dass die Lastkontrollschicht immer individuell abgestimmt werden muss. Sie funktioniert nur dann gut, wenn die Statements des Workloads bekannt sind (damit man weiß, wo und wie einzugreifen ist) und der Workload selbst gleichbleibend ist. Wechselt der Workload häufig, so müsste auch das Regel-File ständig angepasst werden.

- ▷ Die RMF selbst verbraucht ebenfalls Ressourcen und erzeugt Last auf dem z/OS-System. Zwar ist diese Last sehr gering (die RMF verbraucht ca. 0.9% der CPU), aber trotz allem sollte man diesen Faktor berücksichtigen.

13. Ausblicke

Das implementierte Lastkontrollsystem ist ein Prototyp - wenn auch ein durchaus *nutzbarer*. Einige Verbesserungen und Erweiterungen sind jedoch denkbar, die im Folgenden kurz umrissen werden. Hierbei soll sowohl auf Verbesserungen als auch Erweiterungen und neue Nutzungsmöglichkeiten eingegangen werden.

- ▷ Auf Grundlage der automatisierten Tests ist auch eine **automatisierte Anpassung der Regeln** denkbar. Da mit der Standard Abweichung eine Maßzahl für die Güte der Ergebnisse zur Verfügung steht, sind Tools vorstellbar, die die Regeln solange anpassen, bis die Abweichung minimal ist.

- ▷ Der **Regelmechanismus könnte erweitert werden**, um beispielsweise Vererbung oder `or`-verknüpfte Bedingungen zuzulassen. So würde das Schreiben von *großen* Regeldateien erleichtert werden.

- ▷ Eine **allgemeine Schnittstelle für Performance-Daten** würde die Abhängigkeit von der RMF verringern. Das Lastkontrollsystem ist in seiner jetzigen Form auf sie angewiesen - was für die behandelte Problemstellung durchaus zielführend war. Würde eine generelle Schnittstelle existieren, um beliebige neue Performance-Daten anschließen zu können, könnte das Lastkontrollsystem nicht nur mit der RMF und z/OS funktionieren, sondern mit jedem Betriebssystem bzw. jeder Datenbank, für die Leistungsdaten abgefragt und der Schnittstelle übergeben werden können.

- ▷ Das Prinzip des Lastkontrollsystems kann **nicht nur auf Datenbanken** angewendet werden. Es würde sich ohne Änderungen beispielsweise auch auf einen Webserver (auf z/OS) anpassen lassen (z.B. den WebSphere Application Server for z/OS). In diesem Fall könnte statt auf den JDBC-Treiber etwa auf Klassen des Packages `java.net` Einfluss genommen werden, die für HTTP-Verbindungen zum Server genutzt werden. Lediglich das *Regel-File* wäre anders.

Literaturverzeichnis

Apache. *Apache IO 2.0.1 API.* Apache Software Foundation, 2010. URL `http://commons.apache.org/io/api-2.0.1/index.html`.

Pierre Cassier, Raimon Korhonen, Peter Mailand, and Michael Teufel. *Effective zSeries Performance Monitoring Using Resource Measurment Facility.* IBM, 2005.

Karl Eilbrecht and Gernot Strake. *Patterns Kompakt.* Spektrum Verlag, 2010.

Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture.* Addison Wesley, 2002.

Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell.* O'Reilly, 2003.

IBM. Developing with jaxb, 02 2008. URL `http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.wasfpws_v6/wasfpws/6.1/JAXB.html`.

IBM. *Resource Measurement Facility Programmer's Guide.* IBM, 2011a.

IBM. *Report Reference.* IBM, 2011b.

Walter Kriha and Roland Schmitz. *Internet-Security aus Software-Sicht.* Springer Verlag, 2008.

R.J. Lorimer. Instrumentation: Modify applications with java 5 class file transformations, 06 2005. URL `http://www.javalobby.org/java/forums/t19309.html`.

Cary Millsap. Thinking clearly about performance, 5 2010. URL `http://method-r.com/downloads/doc_download/44-thinking-clearly-about-performance`.

Ramesh Natarajan. 10 useful sar (sysstat) examples for unix / linux performance monitoring, 2011. URL `http://www.thegeekstuff.com/2011/03/sar-examples/`.

Uwe Schneider and Dieter Werner. *Taschenbuch der Informatik.* Carl Hanser Verlag, 2007.

Dennis Sosnoski. Java programming dynamics, part 4: Class transformation with javassist, 09 2003. URL `http://www.ibm.com/developerworks/java/library/j-dyn0916/index.html`.

Thomas Uhrig. Praxissemester bei ibm, 2011. URL `http://tuhrig.de/wp-content/uploads/Praxisbericht.pdf`.

Christian Ullenboom. Java ist auch eine insel, 2011. URL `http://openbook.galileocomputing.de/javainsel/`.

Dr.-Ing. Ning Wu. Verkehr auf schnellstraßen im fundamentaldiagramm, 2000. URL `http://homepage.ruhr-uni-bochum.de/Ning.Wu/pdf/FMDG_SVT_8_2000.pdf`.

A. Anhang

Wichtige Leistungsindikatoren mit RMF-Bezeichnern

	Erklärung	**RMF Ressource**	**RMF ID**	**RMF Name**
Prozentuale Auslastung der CPU pro Datenbank	Gibt wieder, wie stark die jeweiligen Datenbanken die CPU belasten. Die CPU-Auslastung ist ein wichtiger Indikator für die Leistung des Systems.	RSE1,*,PROCESSOR	8D27B0	% eappl by WLM report class
Beendete Transaktionen pro Zeiteinheit	Kein direkter Performance-Wert einer Ressource. Hat man aber stets mit ähnlichen Transaktionen zu tun, so kann über diesen Wert auf andere konkrete Auslastungen geschlossen werden.	,RSEPLEX,SYSPLEX	8D1220	transaction ended rate by WLM report class
Gesamtauslastung der CPU	Gibt wider, wie stark die CPU insgesamt ausgelastet ist. Hier fallen auch andere Datenbanken und Anwendungen ins Gewicht.	,RSE1,MVS_IMAGE	8D0450	% processor utilization by MVS image
Prozentuale Zeit, die die Datenbank während des Messintervalls auf die CPU warten musste	Je höher dieser Wert, desto häufiger war die CPU beschäftigt und nicht verfügbar. Ist die Zeit groß, so muss die Datenbank häufig und lange auf die CPU warten.	RSE1,*,PROCESSOR	8D16A0	% delay by WLM report class
Gesamte Auslastung des zIIP-Prozessors	Gibt wieder, wie stark der zIIP-Prozessor ausgelastet ist. Dieser Prozessor muss vom Kunden nicht pro Takt bezahlt werden.	RSE1,*,PROCESSOR	8D39C0	% CPU utilization (IIP)
Prozentuale Auslastung des zIIP-Prozessors pro Datenbank	Gibt wieder, wie stark der zIIP-Prozessor von der jeweiligen Datenbank ausgelastet ist. Dieser Prozessor muss vom Kunden nicht pro Takt bezahlt werden.	RSE1,*,PROCESSOR	8D34F0	% IIP by WLM report class

Tabelle A.1.: Wichtige Leistungsindikatoren (mit RMF-Bezeichnern)

Tabelle der verwendeten Bibliotheken (zusätzlich zu Java 6)

Bibliothek	Version	Verwendung	Webseite
Apache Commons IO	2.0.1	Zur Überwachung des Regel-Files auf Änderungen.	http://commons.apache.org/io
Apache Commons Math	2.2	Zur Berechnung von Mittelwerten, Standartabweichung und ähnlichem.	http://commons.apache.org/math
Apache ANT	1.0b2	Zum Bau der Applikation und automatisierten Tests.	http://ant.apache.org
ANT Contrib	1.8.2	Erweiterung, um try-catch-Blöcke in ANT zu nutzen.	http://ant-contrib.sourceforge.net
Apache Log4j	1.2.16	Zum Logging.	http://logging.apache.org/log4j
JBoss Javassist	3.11	Zur Bytecode-Manipulation.	http://www.jboss.org/javassist
BIRT	2.6	Zur automatischen Generierung von Charts in den Tests.	http://eclipse.org/birt/phoenix
JAXB	2.0	Zur Generierung von Java-Klassen aus XSD-Dateien.	-
Test Applikation der IBM		Ein IBM-internes Programm, mit dem die Lastkontrolle getestet wurde.	-

Tabelle A.2.: Verwendete Bibliotheken