Skalierungskonzepte in datenintensiven, mehrschichtigen und verteilten Anwendungsarchitekturen und ihre quantitative Bewertung.

Burkhard Gauß
burkhard@familie-gauss.de

23. Mai 2003

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegeben Literatur angefertigt habe.

Kaiserslautern, den 23. Mai 2003

_____

Burkhard Gauß

Inhaltsverzeichnis

Kapitel 1

Einleitung

In modernen Systemen werden heutzutage immer häufiger mehrschichtige und verteilte Software-Architekturen eingesetzt. Besonders zur Realisierung von datenintensiven Anwendungen, wie etwa Server-Dienste im Internet, werden solche Architekturen bevorzugt. Sie bieten im Vergleich zu herkömmlichen Software-Architekturen in diesem Bereich viele Vorteile:

- Sie erlauben ein einfaches und dynamisches Skalieren.
- Sie bieten eine hohe Ausfallsicherheit.
- Es können viele „Standard“ PC-Systeme „zusammengeschaltet“ werden um eine hohe Gesamtleistung zu erreichen.

Ein weiterer Grund für die Verbreitung mehrschichtiger und verteilter Systeme sind die inzwischen zur Verfügung stehenden Laufzeitumgebungen und deren Komponentenmodelle. So stellen beispielsweise Microsofts .NET [Mic03a] oder Suns Java 2 Enterprise Edition [Mic03b] jeweils eine umfangreiche Sammlung von Konzepten, Modellen und Diensten zur Verfügung um ein verteiltes, mehrschichtiges System zu entwickeln und betreiben zu können. Vereinfacht kann man sagen, diese Laufzeitumgebungen verbergen einen großen Teil der „Mehrschichtigkeit“ und der „Verteilung“ vor den Entwicklern und erleichtern so den Umgang mit diesen Techniken enorm.

Gleichzeitig hat mit der Einführung dieser Laufzeitumgebungen auch die Bedeutung komponentenbasierter Software-Systeme stark zugenommen. In diesem Zusammenhang tritt immer wieder die Frage auf, ob eine Komponente interne Zustände bezüglich ihrer Clients (Conversational States) verwalten soll oder nicht. Verwalten die Instanzen einer Komponente interne Zustände, etwa indem Ergebnisse von Berechnungen gespeichert werden, so sind diese Instanzen mehr oder weniger exklusiv an ihre Client-Instanz gebunden. Verlagert man die Zustandsverwaltung allerdings in die aufrufende Komponente (oder Client-Instanz) so entfällt die exklusive Bindung zwischen den Beiden. Die Instanzen solcher zustandsloser Komponenten können somit von verschiedenen Clients nacheinander verwendet werden.

In dieser Diplomarbeit werden diese beiden Implementierungs- bzw. Entwurfskonzepte hinsichtlich ihrer Fähigkeit zu skalieren untersucht. Beide Konzepte bieten diesbezüglich Vor- und Nachteile. Die zustandslosen Komponenten lassen ein gutes Skalierungsverhalten erwarten: Es müssen immer nur so viele Instanzen einer Komponente erzeugt werden, wie parallel zum Einsatz kommen. inaktive Instanzen gibt es keine (bzw. sehr wenige, etwa in einem Pool). Der Nachteil dieses Entwurfskonzeptes ist der hohe Kommunikationsaufwand zum Austausch der Zustandsinformationen zwischen aufrufender und aufgerufener Instanz.

Diese Problematik existiert bei einem Komponentenmodell mit zustandsbehafteten Komponenten nicht. In jeder Instanz können die benötigten Zustandsinformationen direkt gespeichert werden. Durch die exklusive Bindung einer Instanz an ihren Erzeuger, muss allerdings eine viel größere Anzahl von Instanzen erzeugt und verwaltet werden: Für jeden Client wird eine eigene Instanz erzeugt, die Verwaltung der Instanzen in einem Pool ist nicht möglich.

Der praktische Teil dieser Diplomarbeit wurde im Rahmen des Meta-Akad Forschungsprojekts (siehe [Fle02] und [Gei00]) erstellt. Das Meta-Akad Projekt besteht in Kern aus der Entwicklung eines Systems zum Auffinden und Verwalten von elektronisch verfügbaren Lehr- und Lernmaterialien. Dies beinhaltet einerseits die technische Entwicklung eines geeigneten Softwaresystems und andererseits das Sammeln und Aufbereiten der Lehr- und Lernmaterialien.

Um das Skalierungsverhalten zustandsloser und zustandsbehafteter Komponenten evaluieren zu können, wurden mehrere unterschiedliche Implementierungen der Resultset-Prozessor Komponente im Meta-Akad Projekt vorgenommen. Mittels eines Lastgenerators sollen die maximale Anzahl der „Interaktionen“ pro Sekunde und die durchschnittliche Antwortzeit einer Interaktion in Abhängigkeit von der Last für jede dieser Implementierungen bestimmt werden. Da die übrigen Komponenten des Systems hierbei unangetastet bleiben, erlauben die Messergebnisse Rückschlüsse auf das Skalierungsverhalten der unterschiedlichen Implementierungen.

Ziel dieser Diplomarbeit ist es aussagekräftige Ergebnisse über das Skalierungsverhalten von zustandslosen und zustandsbehafteten Komponenten zu erhalten.

Kapitel 2

Gliederung

Im Folgenden wird die Gliederung dieser Diplomarbeit erläutert, wobei für jedes Kapitel eine kurze Inhaltsübersicht gegeben wird.

In Kapitel 3 werden die Grundlagen zum Verständnis mehrschichtiger und verteilter Software-Architektur-Konzepte erläutert. Der erste Abschnitt erklärt die Begriffe „Softwarearchitektur“ und „Verteilte-Software-Systeme“. Dabei werden einige Merkmale von Software-Architekturen vorgestellt, die zur Bewertung der in den folgenden Abschnitten beschriebenen Konzepte herangezogen werden. Ein besonderer Schwerpunkt liegt auf dem Merkmal „Skalierbarkeit“, welches ausführlich beschrieben wird. Im zweiten Abschnitt des Kapitels wird das Client-Server-Konzept in zwei Variationen als Software-Architektur vorgestellt und bewertet. Der dritte Abschnitt beschäftigt sich mit mehrschichtigen Architektur-Konzepten. Er stellt die N-Schicht Architektur vor und bewertet diese anhand der im ersten Abschnitt genannten Merkmale. Abschließend werden einige Probleme die im Zusammenhang mit mehrschichtigen Software-Architekturen auftreten erläutert.

Das 4. Kapitel widmet sich Komponenten und komponentenbasierter Software-Entwicklung. Nach einer Einführung in die Thematik im ersten Abschnitt, wird im zweiten zunächst der Begriff Komponente definiert. Dabei werden verschiedene Definitionen aus der Literatur und dem Internet einander gegenübergestellt. Anschließend wird der Versuch unternommen diese Definitionen in einer Gemeinsamen zusammenzuführen. Der letzte Abschnitt erläutert den Begriff „Zustand“ im Hinblick auf Komponenten bzw. deren Instanzen, dabei werden besonders die Unterschiede zwischen Komponenten mit und ohne Zustand hervorgehoben.

Kapitel 5 bietet eine Einführung in die Java 2 Enterprise Edition. Das erste Teilkapitel beschreibt unter Bezugnahme auf Kapitel 3 das Architektur-Konzept von J2EE. Anschließend wird im zweiten Abschnitt das Komponenten-Modell von J2EE erläutert. Dabei werden die unterschiedlichen Komponenten-Typen der Java 2 Enterprise Laufzeitumgebung erklärt. Besonders die Session-Beans in den Varianten Stateless- und Stateful-Session-Bean werden eingehend betrachtet. Zuletzt werden in diesem Abschnitt

noch die Techniken zur Komposition einzelner J2EE-Komponenten beschrieben.

Das Kapitel 6 beschreibt das Meta-Akad Forschungsprojekt in dessen Rahmen diese Diplomarbeit entstanden ist. Nachdem im ersten Abschnitt eine kurze Übersicht über den Zweck und die Ziele des Projektes gegeben wurde, widmet sich der Zweite den technischen Entwurfskonzepten des Meta-Akad-Systems. In einer Übersicht werden die einzelnen Komponenten des Systems vorgestellt und anschließend ihre Aufgabe erläutert. Am Ende des zweiten Abschnittes wird, um die Funktionsweise des Meta-Akad-Systems zu verdeutlichen, die Abarbeitung einer Anfrage beispielhaft beschrieben. Der dritte Teil widmet sich der Lösung konzeptioneller Probleme im Zusammenhang mit der zustandslosen Implementierung des Resultset-Prozessors.

Der erste Teil des 7. Kapitels beschreibt die internen Abläufe des Query-Prozessors eingehender. Daran anschließend werden im zweiten Teil Implementierungsdetails der verschiedenen Versionen des Resultset-Prozessors erläutert. Dabei werden jeweils explizit die erwarteten Vor- und Nachteile der unterschiedlichen Varianten herausgestellt. Das Kapitel wird durch eine detaillierte Beschreibung der internen Abläufe einer Suchanfrage abgeschlossen.

Das Kapitel 8 stellt die zur Evaluierung eingesetzten Konzepte und deren Umsetzung vor. Der erste Teil des Kapitels beschreibt unter welchen Rahmenbedingungen, welche Werte gemessen werden sollen. Im zweiten Teil wird der entsprechend dieser Rahmenbedingungen entworfene Lastgenerator erläutert.

Im 9. Kapitel wird zunächst die Messumgebung beschrieben in der die im Rahmen dieser Diplomarbeit vorgenommenen Tests abgelaufen sind. Daran anschließend werden die durchgeführten Messreihen erläutert und ihre Ergebnisse ausgewertet und interpretiert.

Zum Ende dieser Diplomarbeit werden in Kapitel 10 die erzielten Ergebnisse nochmals zusammengefasst und ein abschließendes Fazit gezogen.

Kapitel 3

Software Architekturen

Zu Beginn sei ein Überblick über verschiedene Software-Architekturen gegeben. Dabei werden im Besonderen die Aspekte ihrer Skalierbarkeit betrachtet. Eine ausführliche Einleitung in das Themengebiet ist in [BRRJ99] und [Mel03] zu finden.

3.1 Eigenschaften von Software-Architekturen

Bevor die Eigenschaften von Software-Architekturen besprochen werden, muss der Begriff *Software-Architektur* selbst erläutert werden. In ihrem Buch „The UML Modeling Language“ definieren Grady Brooch (u.a.) den Begriff Software-Architektur wie folgt:

„*An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition.*“

Eine Software-Architektur kann somit als Menge von Entscheidungen (oder Vorgaben) über die Organisation einer Software angesehen werden. Zusätzlich werden durch die Architektur die zu verwendenden strukturellen Elemente, die Art der Zusammenarbeit einzelner Elemente, ihre Schnittstellen und ihre Verknüpfungen untereinander festgelegt. Dieser Definition folgend, hat jedes Software-System eine ihm zugrundeliegende Software-Architektur, auch wenn diese häufig implizit durch den Akt das Software-Entwurfs entsteht. Hier sollen allerdings vorgegebene Architektur-Konzepte betrachtet werden, die explizit beim Entwurf eines Software-System zum Einsatz kommen. Die Organisation und der strukturelle Aufbau einer Architektur sind hier besonders unter dem Gesichtspunkt der „Verteilung“ zu betrachten. Die Interfaces, ihre Komposition und ihre Zusammenarbeit werden, bei den im Folgenden betrachteten Architekturen, nicht weiter erläutert.

In den Abschnitten 3.1.3, 3.1.4 und 3.1.5 werden einige Aspekte von Software-

Architekturen erläutert. Die in den daran anschließenden Abschnitten vorgestellten Architektur-Konzepte sollen jeweils anhand dieser Aspekte bewertet werden.

3.1.1 Verteilte Software-Systeme

Verteilte Software-Systeme sind Systeme, die auf mehrere verschiedene Prozesse auf Betriebssystemebene bzw. auf mehrere verschiedene physikalische Systeme verteilt sind.

Entsprechende Entwicklungs- bzw. Laufzeitumgebungen unterstützen dies indem sie eine passende Infrastruktur bereitstellen. Diese stellt wiederum dem Anwendungsprogramm eine Reihe von Diensten zur Verfügung, die den Umgang mit den einzelnen Aspekten der „Verteiltheit“ erleichtern. Eine solche Infrastruktur, häufig auch Framework genannt, ist beispielsweise die J2EE Laufzeitumgebung. Weiteres zu J2EE und dem Begriff Framework findet sich in Kapitel 5.

3.1.2 Datenintensive Software-Systeme

Ein datenintensives Software-System ist ein System in dem große Mengen an Daten gespeichert, verarbeitet und übertragen werden müssen. Besonders bei den immer schneller zunehmenden Datenmengen im Internet sind datenintensive Systeme inzwischen die Regel.

3.1.3 Skalierbarkeit von Software-Systemen

Zentraler Aspekt unter dem die folgenden Architektur-Konzepte betrachtet werden sollen, ist ihre Skalierbarkeit. Mit Skalierbarkeit ist das Vermögen eines Software-Systems gemeint, sich dynamisch wechselnden Auslastungen anpassen zu können. Bei wechselnder Systemlast ist allerdings nur der Fall der steigenden Last als kritisch anzusehen. Hierbei können Engpässe auftreten die der Skalierbarkeit des Systems eine obere Grenze setzten. Im Falle sinkender Last ist hingegen nicht mit Problemen zu rechnen, da hier keine Engpässe auftreten können (es werden lediglich Ressourcen freigesetzt). Ein optimal skalierendes System ist nur durch die Hardware auf der es läuft beschränkt. Man unterscheidet drei Arten der Skalierbarkeit: [Mic03e]

vertikale Skalierung Unter vertikaler Skalierung kann man einerseits die Hardware-Erweiterung eines bestehenden Systems verstehen, beispielsweise durch Hinzufügen von zusätzlichen CPUs, Festplatten oder zusätzlichem Hauptspeicher. Andererseits wäre auch die Erweiterung der Leistungsfähigkeit der Software eine Möglichkeit zur vertikalen Skalierung. Viele Software-Produkte enthalten künstliche Beschränkungen bezüglich ihrer Skalierbarkeit, entsprechend der beim Hersteller eingekauften Lizenz. So ist es etwa bei Datenbankverwaltungssystemen (DBVS) üblich die Leistung dahingehend zu limitieren, dass diese nur eine vorgegebene Anzahl von parallelen Verbindungen erlauben. Benötigt man mehr Ressourcen (etwa in Form von Verbindungen) als man ursprünglich kalkuliert hat, so kann man durch Hinzukaufen von weiteren Lizenzen eine vertikale Skalierung des Software-Systems vornehmen. Die vertikale Skalierungstechnik kann bei jeder Software-Architektur

eingesetzt werden. Sie ist vor allem für eine sich langsam und vorhersehbar ändernde Systemlast geeignet.

horizontale Skalierung Man kann einerseits monolithische, nicht verteilte Systeme horizontal skalieren in dem man sie in mehrere Softwaremodule bzw. Teilsysteme aufteilt. Diese Teilsysteme können dann einfach auf verschiedene physikalische Systeme verteilt werden. Dies bedeutet aber, dass man das der Software zugrundeliegende Architektur-Konzept verändert und ist in der Regel mit einem hohen programmiertechnischen Aufwand verbunden. Einfacher lässt sich eine horizontale Skalierung bei „von Haus aus“ verteilten Systemen vornehmen. In diesem Fall skaliert man das System, je nach Bedarf, entweder auf jeder Schicht durch Hinzufügen von zusätzlichen, parallel arbeitenden Servern (mit den entsprechenden Schichten) oder man verteilt Schichten die bislang auf einem gemeinsamen Server liefen auf mehrere Server. Diese Server können einerseits Server in Form von zusätzlicher Hardware sein, sie können aber auch durch eine zusätzliche Instanz der Software auf der bestehenden Hardware dargestellt werden. Um ein solches System zu skalieren, gibt es zwei grundsätzliche Vorgehensweisen, die aber beliebig kombiniert werden können. Der erste Weg ein solches System zu skalieren ist das Verteilen der bestehenden Schichten auf mehrere unterschiedliche Systeme. Der zweite Ansatz ist das Hinzufügen von zusätzlichen parallel arbeitenden Schichten auf dem gleichen oder einem zusätzlichen System. Das Framework von Mehrschichtsystemen bietet in der Regel Möglichkeiten zur dynamischen, und für die anderen Schichten transparenten, Lastverteilung. Um eine horizontale Skalierung vorzunehmen, müssen weder auf Seite der Clients, noch auf Seite des Anwendungscodes, Anpassungen vorgenommen werden. Dadurch ist es möglich eine horizontale Skalierung zur Laufzeit durchzuführen. Sie eignet sich somit besonders zur schnellen, dynamischen Anpassungen der Systemleistung.

diagonale Skalierung Die Kombination der beiden zuvor genannten Skalierungsarten nennt man diagonale Skalierung: Langfristige, vorhersehbare Lastveränderungen werden im bestehenden System horizontal skaliert. Kurzfristige, dynamische Änderungen in der Systemlast können mittels horizontaler Skalierung abgefangen werden.

Der Einsatz horizontaler bzw. diagonaler Skalierungstechniken setzt jedoch in jedem Fall ein Architektur-Konzept vorraus, das verteilte Systeme unterstützt. Ein monolithisches Software-System lässt sich normalerweise nicht, ohne große Anpassungen vornehmen zu müssen, auf mehrere parallele Systeme verteilen.

3.1.4 Ausfallsicherheit

Unter Ausfallsicherheit wird die Frage, wie wahrscheinlich ein Systemausfall ist, verstanden. In verteilten Systemen sind dabei besonders die Auswirkungen auf das Gesamtsystem, bei Ausfall eines Teilsystems zu beachten. Bei verteilten Systemen, kann der Ausfall einer

Abbildung 3.1: Architektur von Client-Server Systemen mit beispielhafter Aufgabenverteilung.

(zentralen) Komponente zu großen Problemen führen, denn meist ist eine sehr große Anzahl von Anwendern betroffen.

3.1.5 Wartungsaufwand

Der Aufwand, den ein Software-System zur Installation und zum Betrieb benötigt, ist unter dem Begriff Wartungsaufwand zusammengefasst. Eine Faustregel besagt, dass der Aufwand den ein Software-System während seines Lebenszyklusses verursacht im Verhältnis 80:20 bezüglich Wartung und Entwicklung steht. Es ist also zu erwarten, dass etwa 80% der Zeit für die Wartung des Systems benötigt wird. Dadurch wird der Wartungsaufwand eines Software-Systems zu einer gewichtigen Größe.

Da diese Diplomarbeit sich vor allem mit Skalierungsaspekten beschäftigt, werden die nachfolgenden Architektur-Konzepte hauptsächlich diesbezüglich bewertet. Die Ausfallsicherheit und der zu erwartende Wartungsaufwand einer Architektur werden jeweils nur kurz angesprochen.

3.2 Client - Server Architekturen

In der Vergangenheit wurden bei verteilten Systemen meist einfache Architekturen nach dem Client-Server Prinzip eingesetzt. Hierbei handelt es sich um verteilte Systeme mit einer 2-Schicht Architektur. Die Abbildung 3.1 stellt eine beispielhafte Client-Server Architektur dar und beschreibt mögliche Aufgaben der einzelnen Schichten im Thin- und Fat-Client Konzept. Es ist dabei zu beachten, dass die Übergänge von einer reinen 2-Schicht Architektur zur N-Schicht Architektur manchmal nicht deutlich zu Tage treten: Würde etwa in Abbildung 3.1 der Server zur Datenhaltung auf eine DBVS zurückgreifen, so müsste man eigentlich von einer drei- bzw. N-Schicht Architektur reden. Echte 2-Schicht Architekturen haben allerdings große Nachteile bezüglich ihrer Ausfallsicherheit, Wartbarkeit und Skalierbarkeit. Im Folgenden werden Architektur-Konzepte mit Thin- und mit Fat-Clients vorgestellt.

3.2.1 Thin-Clients

In einem System mit einer Thin-Client Architektur übernehmen die Clients im Wesentlichen Darstellungsaufgaben. Die gesamte Anwendungslogik läuft auf dem Serversystem. Die Clients, häufig ohne eigenen Festspeicher ausgeführt, laden die benötigte Präsentationslogik über eine Netzwerkverbindung vom Server. Dieser stellt in einem weitgehend

monolithischen System die Anwendung bereit. Dieses Szenario erlaubt es, technisch relativ einfache Client-Systeme einzusetzten, lediglich der Server muss den Anforderungen angepasst werden.

Skalierbarkeit Die schon zuvor erwähnten monolithischen Server-Systeme können meist nur vertikal skalieren: Durch Erweiterung ihrer Hardware (Hauptspeicher, CPUs, etc.), durch vollständigen Austausch des (Hardware-)Systems oder durch „Erweitern“ der Software (hinzukaufen von Lizenzen). Die möglichen Erweiterungen der Hardware eines bestehenden Systems können zusätzlich durch die zugrundeliegende Software-Architektur eingeschränkt werden. So ist ein Hinzufügen von zusätzlichen CPUs nur sinnvoll, wenn das Software-System auf mehrere Prozesse verteilt werden kann bzw. ist. Vertikale Skalierungstechniken sind zwar prinzipiell möglich, sind aber meist nicht durchführbar z.B: Wenn kein Zugriff auf den Quell-Code des Software-Systems besteht und so keine Aufteilung in mehrere Teilsysteme möglich ist. (Dies würde außerdem auch automatisch zu einem N-Schicht System führen.) Am Ende bleibt zur Skalierung solcher Systeme in vielen Fällen als einzige Option ihr Austausch. Dies verursacht allerdings meist erhebliche Kosten. Zusammenfassend lässt sich sagen: Systeme mit Thin-Client Architektur skalieren nur sehr schlecht.

Ausfallsicherheit Die Ausfallsicherheit dieser Systeme ist fast zu 100% von der Ausfallsicherheit des Servers abhängig. Durch die sehr hohen Anschaffungskosten eines leistungsfähigen Server-Systems, stehen häufig keine Ersatz- oder Notfallsysteme zur Verfügung. Der Ausfall des Servers bedeutet immer einen Totalausfall des gesamten Systems.

Wartungsaufwand Die Wartung solcher Systeme gestaltet sich recht einfach. Im günstigsten Fall muss lediglich der Server gewartet werden. Die Clients sind bis auf technische Aspekte (Hardware) wartungsfrei.

3.2.2 Fat-Clients

Systeme dieser Kategorie arbeiten mit einem leicht veränderten Architektur-Konzept. Zwar behält man das Client-Server-Modell bei, doch sind die Clients wesentlich leistungsfähiger als die der Thin-Client-Architektur. So wird es möglich einen Teil der Programmlogik direkt auf dem Client auszuführen, was die Anforderungen an das Server-System erheblich reduziert. Normalerweise wird der clientseitige Anteil der Programmlogik direkt auf dem Client installiert, so dass der Server diesen nicht zur Verfügung stellen muss.

Skalierbarkeit Die in 3.2.1 genannten Probleme bei der Skalierbarkeit solcher Systeme gelten auch hier. Die Möglichkeiten zur Skalierung der Software sind identisch zu denen der Thin-Client Architektur. Die Skalierbarkeit der auf Seite der Hardware verbessert sich allerdings leicht. Da durch Hinzufügen von zusätzlichen Clients die Last des Servers nicht

Abbildung 3.2: Beispielhafte N-Schicht Architektur.

so schnell ansteigt, wie bei der Thin-Client-Architektur, dauert es länger bis der Server den Punkt erreicht, an dem er ausgetauscht werden muss. Die Anforderungen an die Leistungsfähigkeit des Server-Systems sind insgesamt nicht so hoch, wie bei der Thin-Client-Architektur. Dadurch ist es möglich, kleinere und somit auch billigere Server-Systeme einzusetzen. Aus Sicht der verfügbaren Hardware bleibt also nach „oben hin“ ein größerer Spielraum.

Ein neues Problem bezüglich der Skalierbarkeit kann allerdings auf Seite der Clients auftreten: Ändert sich die Last auf den Clients, etwa durch größere Datenmengen oder durch eine neue Anwendungssoftware, so müssen unter Umständen alle Clients ausgetauscht bzw. erweitert werden.

Zusammenfassend skaliert diese Architektur etwas besser als die Vorangegangene, die grundsätzlichen Problematiken (nur vertikale Skalierbarkeit) bleiben allerdings bestehen. Zusätzlich ist anzumerken, dass beide Client-Server-Architekturen keine dynamischen Anpassungen an die System-Last erlauben.

Ausfallsicherheit Die Ausfallsicherheit dieser Systeme kann, im Vergleich zu den voran genannten, lediglich durch den Einsatz von redundanten Systemen erhöht werden. Da diese auch bei der Thin-Client Architektur möglich sind, ergibt sich hier allerdings kein prinzipieller Vorteil. Es handelt sich vielmehr um einen praktischen Vorteil: Dadurch, dass die Server hier tendenziell nicht so leistungsfähig sein müssen als bei der Thin-Client Architektur, sind sie auch in der Anschaffung kostengünstiger. Dies hat wiederum zur Folge, dass die Wahrscheinlichkeit steigt finanzielle Mittel zur Verfügung zu haben um ein zweites Server-System zu beschaffen. Man könnte sagen: Ein höheres Maß an Ausfallsicherheit ist hier mit geringeren Kosten verbunden als bei Systemen mit Thin-Clients.

Wartungsaufwand Der große Nachteil der Fat-Client Systeme ist jedoch ihr hoher Installations- und Wartungsaufwand. Auf jedem Client muss die Applikation einzeln installiert werden. Soll die Applikation geändert werden, muss diese Änderung auf jedem Client „von Hand“ propagiert werden. Dies kann bei einer großen Zahl von Clients zu einem erheblichen Wartungsaufwand führen.

Moderne Architekturen unterstützen deswegen mehrere Abstraktionsschichten und versuchen dadurch die genannten Nachteile zu umgehen.

3.3 Mehrschichtige Software Konzepte

Im Folgenden werden die Konzepte einer modernen, mehrschichtigen Softwarearchitektur erläutert und den traditionellen Client-Server Architekturen gegenübergestellt.

3.3.1 N-Schicht Architektur

Mehrschichtige Architekturen haben zwischen dem Client und dem Server mehrere Abstraktionsschichten, so genannte Tiers. In [Mic03e] wird eine Schicht wie folgt definiert:

„*A Tier is a functionally seperated hardware and software component that performs a specific function.*“

Durch die funktionale Trennung der einzelnen Schichten und deren Kapselung in Komponenten (siehe auch Abschnitt 4) wird es möglich, jede Schicht auf einem eigenen Rechner-System laufen zu lassen. Eine mehrschichtige Software-Architektur muss hierzu Kommunikationsmechanismen definieren, die den Informationsaustausch zwischen den einzelnen Schichten erlauben.

Die Abbildung 3.2 zeigt ein Beispiel einer mehrschichtigen Software-Architektur. Sie hat auf der untersten Schicht einen DB-Server, der alle im System benötigten oder anfallenden Daten speichert. Schicht 2 stellt eine Caching-Schicht dar, die beispielsweise Daten der Datenbank zwischenspeichert. So könnte etwa die Anzahl der DB-Zugriffe reduziert werden. Zusätzlich könnte eine solche Schicht den darüber liegenden Schichten eine gemeinsame Nutzung von DB-Ressourcen, wie etwa DB-Verbindungen, erlauben (Connection Pooling, siehe Kapitel 5). Die Anwendungslogik übernimmt in diesem Beispiel die gesamte Datenverarbeitung, das heißt: Sie enthält die eigentliche Programmlogik. Die vierte Schicht, könnte zur Aufbereitung der Daten zur Präsentation dienen. So könnte die Präsentationslogik die darzustellenden Daten in eine HTML-Repräsentation bringen. Die Präsentation selbst wird dann von der letzten Schicht, den Clients, vorgenommen. Als Clients könnten hier gewöhnliche Web-Browser dienen, die die HTML-Daten anzeigen und Benutzereingaben wieder an die Präsentations-Schicht zurückliefern. Da nicht nur Benutzereingaben, sondern auch Ergebnisse von Berechnungen an tiefer liegende Schichten weitergeleitet werden müssen, ist der Informationsfluss in solchen Systemen immer bidirektional.

Zusätzliche Schichten können in beliebiger Anzahl zu diesem Modell hinzugefügt werden. In der Beispiel-Architektur könnte, in Anwendungsszenarien mit hoher Last, eine Caching-Schicht zwischen der Präsentations- und Anwendungslogik durchaus sinnvoll sein. Dabei ist jedoch zu beachten, dass jede neue Schicht zusätzlichen Kommunikationsaufwand bedeutet. Man muss also einen Mittelweg zwischen der Aufgliederung in Schichten und dem dadurch entstehenden Kommunikationsaufwand finden.

3.3.2 Skalierbarkeit

N-Schicht-Architekturen gehören in der Regel zur Klasse der Verteilten-Systeme. Es ist zwar möglich alle Schichten auf einen Rechner-System laufen zu lassen, durch die Untergliederung in Schichten entsteht aber automatisch eine gewisse Nebenläufigkeit. So führt die Aufteilung des Software-System in mehrere Ausführungs-Komponenten (DBVS, Anwendungslogik, Client) zu einem besseren Skalierungsverhalten, als es monolithische Systeme aufweisen. N-Schicht-Systeme können so, bei Hinzufügen von zusätzlichen CPUs (verti-

kale Skalierung), größere Vorteile erlangen als monolithische Systeme. Zusätzlich ist hier eine horizontale Skalierung leicht zu realisieren. Um die einzelnen Schichten auf mehrere Rechner-Systeme zu verteilen, muss lediglich die Kommunikation zwischen den Schichten angepasst werden. Heutige Mehrschicht-Architekturen unterstützen dies bereits direkt. Der Code der einzelnen Komponenten (Schichten) ist in solchen Systemen meist nicht zu ändern, wenn das System horizontal skaliert. Ein Beispiel hierfür bietet die J2EE Laufzeitumgebung: Hier kann die Kommunikation zwischen den einzelnen Komponenten und somit auch zwischen den einzelnen Schichten durch eine Reihe von Konfigurationsdateien, den Deployment-Deskriptoren (siehe 5.2.3), gesteuert werden. Indem man entsprechende Änderungen an den Deployment-Deskriptoren vornimmt, lässt sich eine Schicht auf ein anderes System auslagern, bzw. eine zusätzliche parallele Schicht in das bestehende System einbinden. Weiteres hierzu ist in Kapitel 5 zu finden.

Mehrschichtige Software-Architekturen haben somit, im Vergleich zu den beiden Client-Server-Architekturen, ein sehr gutes Skalierungsverhalten. Sie können vertikal skalieren, was in jeder Architektur möglich ist, und sie können horizontal skalieren (in einigen Frameworks sogar dynamisch zur Laufzeit).

3.3.3 Ausfallsicherheit

Durch den parallelen Einsatz mehrerer Server lässt sich gleichzeitig auch die Ausfallsicherheit des Gesamtsystems erhöhen. Beim Wegfall eines Servers genügt es meist die Laufzeitumgebung zu informieren. Diese ist dann dafür zuständig die Kommunikationswege entsprechend anzupassen. Die übrigen Komponenten des Systems bemerken davon in der Regel nichts. Das Gesamtsystem bleibt somit in seiner vollen Funktionalität verfügbar, lediglich die maximale Leistungsfähigkeit wird durch den Wegfall eines parallelen Servers beeinträchtigt.

3.3.4 Wartungsaufwand

Auch die Wartbarkeit von N-Schicht Systemen im Vergleich zur Client-Server Umgebung verbessert sich erheblich. Auf Seite der Clients ist es weiterhin möglich, das Thin-Client-Konzept zu verfolgen, das praktisch wartungsfrei ist. Im Beispiel auf Seite 11 wird bereits ein solcher, wartungsfreier Client genannt: Der Web-Browser.

Somit bleiben in mehrschichtigen Software-Systemen meist nur die Systeme zu administrieren, auf denen die unteren Schichten ablaufen (im Beispiel wären das Schicht 1-4). Die Anzahl dieser Systeme ist jedoch klein im Verhältnis zur Anzahl der Clients, so dass sich hier Vorteile gegenüber der Fat-Client-Architektur (wo auch alle Clients regelmäßige Wartung benötigen) ergeben. Ein weiterer Vorteil bei der Wartung liegt im modularen Aufbau von N-Schicht Systemen (siehe Kapitel 4). Dieser ermöglicht es, einzelne Softwarekomponenten auszutauschen, ohne das gesamte System anpassen zu müssen.

3.3.5 Probleme mehrschichtiger Architekturen

Problematisch kann bei mehrschichtigen Architekturen, wie bereits zuvor angesprochen, besonders die Anzahl der Schichten werden. Zwischen zwei benachbarten Schichten müssen alle Daten, bzw. Informationen über die gegebenen Kommunikationskanäle übermittelt werden. Im Vergleich zu einem monolithischen System bedeutet dies zunächst einmal einen Mehraufwand. Die in den vorangegangenen Abschnitten beschriebenen Vorteile von N-Schicht-Systemen überwiegen allerdings in der Regel. Steigt die Zahl der Schichten, so kann der Mehraufwand für die Kommunikation in der Summe größer werden, als der Gewinn durch die Aufgliederung in Schichten. Beim Einfügen einer zusätzlichen Schicht muss also immer eine Abwägung zwischen den Kosten (zur Kommunikation) und dem Nutzen der neuen Schicht stattfinden.

Kapitel 4

Komponenten

Komponentenbasierte Systeme stellen im Zusammenspiel mit einer mehrschichtigen Software-Architektur eine äußerst flexible Entwurfsumgebung dar. Begriffe wie Komponente, Laufzeitumgebung oder Framework sind im vorangegangenen Kapitel bereits mehrfach gefallen. Nun sollen sie präzisiert und eingehend erläutert werden. Ausführliche Informationen zur komponentenbasierten Software-Entwicklung sind in [Szy98] und [Gri98] zu finden. Anschließend werden im Kapitel 5 die hier vorgestellten Konzepte in das Java 2 Enterprise Framework eingeordnet.

4.1 Komponentenbasierte Software-Entwicklung

Ursprünglich bedeutete Software-Entwicklung, dass ein mehr oder weniger monolithisches Software-System entwickelt wurde. Ein solches System hat in der Regel alle benötigten Funktionalitäten selbst implementiert. Das hatte zur Folge, dass ein großer Teil an Standard-Funktionalität immer wieder implementiert werden musste. Deswegen wurden in der Folgezeit Software-Bibliotheken geschaffen, die Lösungen für Standard-Aufgaben bereitstellten. So konnte die Entwicklung neuer Software-Systeme erheblich vereinfacht und beschleunigt werden. Problematisch war allerdings die nicht, oder nur wenig existierende Trennung der einzelnen Funktions-Blöcke. Dadurch konnten bei „unsauberer“ Programmierung unerwünschte Effekte (Seiteneffekte) auftreten. Diese Problematik konnte durch die Einführung „Objekt-Orientierter“ Programmiersprachen weitgehend entschärft werden. In OO-Systemen wird versucht, Funktionalitäten so zusammenzufassen, dass logische Einheiten entstehen. Diese Einheiten, Klassen genannt, sollten so konzipiert werden, dass zusammengehörige Funktionalitäten eine gemeinsame Klasse bilden. Durch das Gliedern der Funktionalitäten in unterschiedliche Klassen, vereinfacht sich die Handhabbarkeit großer Systeme enorm. Eine Klasse stellt allerdings nicht nur Funktionalitäten zur Verfügung, sondern bietet auch die Möglichkeit Daten in ihr (bzw. ihren Instanzen, den Objekten) abzuspeichern. Da es sich bei einer Klasse um ein abstraktes Konstrukt handelt, muss sie, bevor sie verwendet werden kann, instantiiert werden. Eine solche Instanz einer Klasse nennt man Objekt. Eine Klasse ist beliebig oft instantiierbar, d.h. es können beliebig

viele Objekte einer Klasse instantiiert werden. Jedes Objekt bildet somit eine abgeschlossenen Einheit, die Daten enthalten kann und/oder Funktionalitäten bereitstellt, um (diese) Daten zu verarbeiten. Die Abgeschlossenheit der Objekte verhindert unerwünschte Seiteneffekte im Zusammenspiel mit anderen Objekten. Dadurch wird die Wiederverwendung von Klassen (bzw. ihren Objekten) stark vereinfacht.

In OO-Systemen besteht meist die Möglichkeit, Schnittstellen zu definieren. Implementiert eine Klasse eine bestimmte Schnittstelle (Interface), so lässt sie sich durch jede andere Klasse austauschen, die ebenfalls diese Schnittstelle implementiert.

Diese Eigenschaften erlauben eine schnelle und effiziente Software-Entwicklung. Es handelt sich hierbei allerdings ausschließlich um Konzepte und Techniken zur Programmierung. Nach dem Übersetzen des Programmcodes in ein ablauffähiges Programm sind sie nur noch von untergeordneter Bedeutung. Der Komponenten-Ansatz versucht nun OO-Techniken und OO-Konzepte in die Welt der ausführbaren Systeme zu übertragen. Wie im Folgenden Abschnitt zu sehen sein wird, hat eine Komponente sehr ähnliche Eigenschaften wie die Klassen von OO-Systemen. Der große Unterschied ist, dass eine Komponente eine ausführbare Einheit ist. Das Zusammensetzten von mehreren Komponenten zu einem größeren System geschieht also nicht mehr vor dem Übersetzen, sondern erst danach. Somit werden die aus der OO-Welt bekannten Konzepte, wie Wiederverwendung, Austauschbarkeit, Kapselung, etc. auf die Ebene der übersetzten Programme, oder Programm-Fragmente übertragen. Aus OO-Sicht können Komponenten als Obermenge der Klassen angesehen werden. Jede Klasse ist oder kann eine Komponente sein. Bei OO-Systemen bestehen Komponenten im Allgemeinen aus mehreren Klassen, man kann sagen: Sie sind gröber granuliert als die Klassen. Umgekehrt sind aus Sicht des Komponenten-Nutzers die zu ihrer Entwicklung eingesetzten Techniken und Konzepte nicht von Bedeutung. Für den Einsatz einer Komponente ist es gleichgültig wie die Komponente entwickelt wurde, lediglich die Einhaltung der vorgegebenen Standards für die Schnittstellen etc. ist von Bedeutung. Für den Nutzer der Komponente ist sie lediglich eine ausführbare Einheit deren innere Abläufe ihm verborgen bleiben. Eine ausführliche Definition des Begriffs Komponente folgt im nächsten Abschnitt. Die hier nur grob umrissenen OO-Konzepte können in [BB92] nachgelesen werden.

4.2 Komponente - ein Definitionsversuch

Hier soll geklärt werden, was unter dem Begriff Komponente zu verstehen ist und was die wichtigsten Eigenschaften einer Komponente sind. In der Literatur und im Internet sind hierzu etliche Definitionen zu finden. Unter [Wik03] werden verschiedene Ansätze diskutiert. Hier eine Auswahl der dort vorgeschlagenen Definitionen:

- „*Software components enable practical reuse of software parts and amortization of investments over multiple applications. There are other units of reuse, such as source code libraries, design, or architectures. Therefore, to be specific, software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.*“ (Clemens Szyperski in [Szy98])

- „*A business component represents the software implementation of an autonomous business concept or business process. It consists of the software artifacts necessary to express, implement, and deploy the concept as a reusable element of a larger business system.*“ (Wojtek Kozaczynski in [Koz99])

- „*Components are self-contained instances of abstract data types (ADTs) that can be plugged together to form complete applications.*“ (Dough Schmidt in [Sch99])

Es gibt viele weitere Definitionen für den Komponentenbegriff. Die drei hier vorgestellten Definitionen nennen jeweils wichtige Eigenschaften einer Komponente. Die erste Definition (von Clemens Szyperski) legt besonderen Wert auf die Wiederverwendbarkeit der Komponenten. Mit Wiederverwendbarkeit ist die Möglichkeit, die Komponente in mehreren unterschiedlichen Software-Systemen einsetzten zu können, gemeint. Dies ist ein grundlegendes Merkmal einer Komponente, bzw. ist eines der Hauptziele beim Einsatz von komponentenbasierten Systemen. Zusätzlich stellt Szyperski explizit die „Binary-Unit“-Eigenschaft von Komponenten heraus. Das bedeutet, Komponenten werden erst *nach* dem Übersetzen des Quellcodes in eine Binärform (bzw. Bytecode bei Java) zu größeren Systemen zusammengesetzt. Dies ist ein entscheidender Unterschied zur Objekt-Orientierung und Klassen. Diese werden *vor* dem Übersetzten zu größeren Systemen zusammengefügt.

In der zweiten Definition (von Wojtek Kozaczynski) wird die grobe Granularität der Komponenten besonders hervorgehoben. Dieser Definition folgend, soll eine Komponente ein „autonomes Geschäfts-Konzept“ bzw. einen „Geschäftsprozess“ umfassen. Unter dem Begriff Geschäfts-Konzept bzw. Geschäftsprozess kann ein, möglichst in sich abgeschlossener, Vorgang im Ablauf der Datenverarbeitung verstanden werden. Bei einem solchen Vorgang handelt es sich allerdings nicht um einen einzelnen Algorithmus oder eine Funktion, sondern um einen erheblich größeren Komplex. Um die „Größe“ einer Komponente besser verdeutlichen zu können, ein kleines Beispiel: Wie Kapitel 6 zeigen wird, ist zum Beispiel das Meta-Akad-System in mehrere Komponenten unterteilt. Eine dieser Komponenten ist der RSPController. Er umfasst alle Funktionalitäten zur Verarbeitung einer Menge von Suchergebnissen. Dies sind unter anderem Methoden zum Sortieren, Aufbereiten, Löschen und Abfragen der Ergebnisse. Er umfasst die gesamte Antwort-Verarbeitung im Meta-Akad-System.

Die dritte Definition für den Komponentenbegriff (von Doug Schmidt) legt besonderen Wert auf die Abgeschlossenheit der Komponenten. Damit soll verdeutlicht werden, dass Komponenten eine möglichst geringe Kopplung untereinander aufweisen sollten. Je höher die Kopplung, also je größer die Abhängigkeiten zwischen unterschiedlichen Komponenten sind, um so schwieriger ist eine Wiederverwendung der einzelnen Komponenten.

Die Unterschiede in den genannten Definitionen liegen in vielen Punkten lediglich in der Gewichtung der einzelnen Forderungen. Beispielsweise stellt die Definition von Schmidt, wie bereits erläutert, die Forderung nach Abgeschlossenheit (self-contained) besonders heraus. Aber auch in der Definition von Kozaczynski lässt sich eine solche Forderung erkennen: Der Ausdruck „autonomous business concept or business process“ impliziert in gewissem

Maße eine Abgeschlossenheit der Komponente. Auch in der Definition von Szyperski ist die Forderung nach abgeschlossenen Komponenten in dem Ausdruck „binary units of independent production“ ebenfalls enthalten.
Im Folgenden werden die Aussagen der drei Definitionen zusammengefasst und dabei die wichtigsten Eigenschaften von Komponenten explizit aufgelistet.

1. Komponenten sind binäre Software-Artefakte. Zum Zeitpunkt ihrer Komposition liegen sie als übersetzte Software-Module vor.

2. Komponenten sind grob granuliert. Sie umfassen immer einen größeren Funktionskomplex.

3. Eine Komponente enthält immer nur die Elemente höchstens eines Vorgangs bzw. Geschäfts-Konzeptes. Die Abläufe verschiedener Konzepte sollten nicht vermischt werden.

4. Komponenten sind in sich abgeschlossene Einheiten. Sie haben möglichst wenige Abhängigkeitsbeziehungen zu anderen Komponenten. Das heißt, sie besitzen eine geringe Kopplung.

5. Komponenten werden speziell im Hinblick auf ihre Wiederverwendbarkeit entworfen.

6. Das Verhalten von Komponenten kann zum Zeitpunkt der Verwendung konfiguriert werden. Änderungen am Quellcode (incl. neu Übersetzen) sind nicht nötig.

7. Komponenten kommunizieren ausschließlich über explizit definierte Schnittstellen miteinander.

Der letzte Punkt wurde hinzugenommen, um die Austauschbarkeit der Komponenten zu verdeutlichen: Da die Komponenten nur über die definierten Schnittstellen kommunizieren, lässt sich eine Komponente problemlos durch eine Andere ersetzen. Die neue Komponente muss lediglich die gleichen Schnittstellen bereitstellen wie die Ursprüngliche. Die Komponente selbst erscheint dem Benutzer als Black-Box mit der er über feste Schnittstellen kommunizieren kann. In diesem Zusammenhang spielt auch die Forderung nach einer geringen Kopplung eine große Rolle, denn beim Austausch einer Komponente müssen alle Abhängigkeiten zwischen dieser Komponente und anderen beachtet werden. Punkt sechs erlaubt es beispielsweise einem Software-Produzenten eine Komponente zu verkaufen ohne dem Kunden Einblick in den Quellcode erlauben zu müssen. In nicht komponentenbasierten Systemen ist häufig der Quellcode nötig um Elemente/Strukturen an eigene Anforderungen anpassen zu können.

Viele Autoren (z.B.: Clemens Szyperski in [Szy98]) fordern zusätzlich noch, dass Komponenten keinen Zustand haben dürfen, d.h.: Alle instantiierten Komponenten sind identisch, es werden keine Daten bzw. Statusinformationen innerhalb der Komponenten abgelegt. Eine solche Komponente hat, auch wenn ihre Bezeichnung anderes vermuten lässt,

einen Zustand. Dieser Zustand, der Initialzustand, wird allerdings nie verlassen. Die Komponente befindet sich nach der Abarbeitung einer Methode wieder in ihrem Initialzustand. Die Bewertung dieser Aussage ist Teil dieser Diplomarbeit. Eine ausführliche Beschreibung des Zustandsbegriffs und seine Bedeutung bezüglich komponentenbasierte Software findet sich in Abschnitt 4.4.

4.3 Laufzeitumgebung

Zu einer Komponente gehört immer eine passende Laufzeitumgebung, die die notwendigen Infrastruktur-Dienste zur Verfügung stellt. Im einfachsten Fall handelt es sich hierbei um entsprechend der Bedürfnisse zur Komposition der einzelnen Komponenten entwickelten Code, auch Glue-Code genannt. Meist steht jedoch eine fertige Infrastruktur, wie etwa ein J2EE Applikations-Server, zur Verfügung. Die Laufzeitumgebung der Komponenten ist dann Teil einer solchen Infrastruktur (Framework). Das Framework ist für die Komposition der einzelnen Komponenten verantwortlich, es stellt die zur Kommunikation zwischen den Komponenten nötige Funktionalität zur Verfügung und es ist verantwortlich für die Erzeugung und Verwaltung von Instanzen der Komponenten. Was das Framework im Einzelnen leistet, hängt jedoch stark von der jeweils zugrunde liegenden Technologie ab. So stellt die *Java 2 Enterprise Edition*, wie in Kapitel 5 zu sehen sein wird, viele weitergehende Dienste zur Verfügung. Etliche heute verwendete Frameworks bieten auch die Möglichkeit, zustandsbehaftete Komponenten einzusetzen. Näheres zu Komponenten mit und ohne Zustand findet sich folgenden Abschnitt.

4.4 Zustände von Komponenten

In diesem Abschnitt soll zunächst der Zustandsbegriff im Allgemeinen erläutert werden. Daran anschließend soll genauer erläutert werden, was unter zustandslosen Komponenten bzw. Komponenten mit Zustand zu verstehen ist.

4.4.1 Definition Zustand

In Objekt-Orientierten Systemen ist der Zustand eines Objektes als „die Belegung seiner Attribute“ definiert. Überträgt man diese Definition auf komponentenbasierte Systeme, so erhält man folgende Definition: „Der Zustand einer Komponente ist die Menge der Belegungen aller Objekte innerhalb der Komponente.“ Da die Definition der Komponente keine keine bestimmte Programmiertechnik für die Entwicklung vorschreibt, ist diese Definition des Zustandes zu einschränkend. Eine allgemeinere Form dieser Definition ist: „Der Zustand einer Komponente ist die Menge alle in ihr gespeicherten Informationen.“

Eine Komponente die, aus Sicht ihrer Clients, unterschiedliche Zustände annehmen kann, hat somit die Möglichkeit Informationen intern zu speichern. Dadurch ist es möglich, dass das Ergebnis einer Berechnung (oder Methodenaufrufs) von den Ergebnissen vorangegangener Berechnungen abhängig ist. Nicht nur die Ergebnisse von unterschiedlichen Me-

thodenaufrufen können den Zustand einer Komponente ändern, auch die Reihenfolge von Methodenaufrufen kann sich in einem bestimmten Zustand niederschlagen. Dies erlaubt es, Ergebnisse einer Berechnung über mehrere Methodenaufrufe hinweg verwenden zu können, ohne dass diese ständig neu berechnet oder außerhalb der Komponente gespeichert werden müssten.

Die Möglichkeit, Zustände von Komponenten verwalten zu können, bringt verschiedene Vorteile, aber auch Nachteile mit sich. Im Folgenden werden die Vor- und Nachteile von Komponenten mit und ohne Zustand erläutert.

4.4.2 Zustandslose Komponenten

Die Instanzen zustandsloser Komponenten besitzen, wie ihr Name schon sagt, keine Zustände oder Daten. Eine solche Komponente kann ihre Dienste somit mehreren verschiedenen Clients zur Verfügung stellen, ohne dass Konflikte auftreten. Die Verwaltung der einzelnen Komponenten durch das zugrundeliegende Framework wird dadurch stark vereinfacht. Jeder Client der eine Komponente verwenden möchte, fordert eine Instanz bei der Laufzeitumgebung an. Im Falle zustandsloser Komponenten muss die Laufzeitumgebung die angeforderte Komponente bzw. eine ihrer Instanzen erst dann dem Client bereitstellen, wenn eine Methode darin aufgerufen wird. Bei dieser Instanz ist es allerdings unerheblich, ob sie zuvor bereits von einem anderen Client verwendet wurde oder nicht. Die Instanz einer Komponente speichert schließlich keine Informationen und kann somit nacheinander von verschiedenen Clients verwendet werden. Die unterschiedlichen Clients verwenden die von ihnen angeforderten Instanzen nicht permanent und auch nicht immer zeitgleich. Dies erlaubt es der Laufzeitumgebung weniger Instanzen einer Komponente bereitzustellen, als in der Summe von den Clients angefordert wurden. Die so entstehende Ressourceneinsparung lässt auf eine bessere Skalierbarkeit des Systems unter hoher Last hoffen. Allerdings muss jede Methode einer zustandslosen Komponente die einen bestimmten Kontext zur Ausführung benötigt diesen vor ihrer Ausführung zunächst herstellen. Kommt es im Ablauf der Methode zu Änderungen im Kontext der Komponente (Zustandsänderungen) die möglicherweise in späteren Methodenaufrufen benötigt werden, so muss dieser Kontext gesichert werden. Eine einfache Art dies zu erreichen ist die Speicherung aller benötigten Zustandsinformationen im Client der Komponente. Dieser kann beispielsweise bei jedem Methodenaufruf die benötigten Zustandsinformationen als Parameter mit übergeben. Umgekehrt können Zustandsänderungen im Rückgabewert der Methode an den Client zurückgeliefert werden. In jedem Fall bedeutet dies aber einen Mehraufwand, welcher einen Teil der zuvor genannten Resourceneinsparungen wieder zunichte macht.

4.4.3 Komponenten mit Zustand

Instanzen von Komponenten mit Zustand[1] können im Gegensatz dazu Zustände bzw. Daten über mehrere Methodenaufrufe hinweg speichern. So wird es möglich, innerhalb

[1] Wie schon gesehen, hat jede Komponente einen Zustand. Im Folgenden werden hiermit Komponenten, die aus Client-Sicht unterschiedliche Zustände annehmen können, bezeichnet.

einer Instanz Ergebnisse einer Berechnung zu speichern. Diese Ergebnisse können dann bei späteren Berechnungen wiederverwendet werden. Es kann somit eine wiederholte Berechnung von Ergebnissen vermieden werden. Dadurch ist es nötig, jedem Client seine eigene Komponenten-Instanz zuzuordnen. Die zugrundeliegende Laufzeitumgebung ist dafür verantwortlich, dass jeder Client nur mit der ihm zugeordneten Instanz kommuniziert. Es müssen also immer genau so viele Instanzen zur Verfügung gestellt und verwaltet werden, wie von den Clients angefordert wurden. Insgesamt lässt dies einen erheblich höheren Ressourcenbedarf auf Serverseite erwarten. Hier entfällt allerdings das aufwendige Sichern und Wiederherstellen der benötigten Zustandsinformationen bei jedem Methodenaufruf, was sich wiederum positiv auf das Skalierungsverhalten auswirkt.

Beide Konzepte bieten somit unterschiedliche Vor- und Nachteile bezüglich ihrer Skalierbarkeit. In der Summe lässt sich an diese Stelle nur schwer vorhersagen, welches dieser beiden Komponenten-Konzepte ein besseres Skalierungsverhalten zeigen wird.

Kapitel 5

Java 2 Enterprise Edition

Diese Kapitel soll eine kurze Einführung in die Software-Architektur und das Komponenten-Modell der Java 2 Enterprise Edition (J2EE) geben. Der Schwerpunkt in den folgenden Erläuterungen wird jeweils auf die im Meta-Akad-Projekt eingesetzten Konzepte und Techniken gelegt. Eine Einführung, sowie weitergehende Dokumentationen zur Java 2 Enterprise Edition liefern [Mic03b], [Rom99] und [SSJtET02].

5.1 N-Schicht-Konzept der Java 2 Enterprise Edition

Der Java 2 Enterprise Edition Laufzeitumgebung liegt eine mehrschichtige Software-Architektur, wie sie in Kapitel 3 beschrieben wurde, zugrunde. Die Abbildung 5.1 zeigt das Grundmodell einer J2EE Architektur. Zusätzliche Schichten können allerdings bei Bedarf hinzugefügt werden. Die dargestellte Einteilung in die Schichten Client-Tier, Middle-Tier und EIS-Tier wird von Sun Microsystems vorgeschlagen. Man könnte hier allerdings auch von einer vier Schicht-Architektur reden, wenn man den Web-Container und den EJB-Container jeweils als eine eigene Schicht ansieht. Man spricht in diesem Zusammenhang von einem Web-Tier und einem EJB-Tier. Die Bedeutung dieser beiden Schichten, bzw. der beiden Container wird in Abschnitt 5.2 genauer erläutert. Die genaue Aufteilung in die unterschiedlichen Schichten, bzw. die Festlegung, welche der gegebenen Schichten des Grundmodells verwendet werden sollen, wird in jedem Fall erst beim Entwurf eines Software-System vorgenommen.

Die oberste Schicht, Client-Tier genannt, ist die Ausführungsschicht der einzelnen Clients. J2EE kennt mehrere unterschiedliche Arten von Clients:

Web-Client Ein Web-Client, etwa ein Internet-Browser, kommuniziert über eine HTTP Verbindung mit einem Web-Container. Der Browser dient hierbei fast ausschließlich zur

Abbildung 5.1: Grundmodell einer mehrschichtigen J2EE Architektur.

Darstellung der gelieferten Daten. Lediglich einfache Funktionalitäten, etwa zur Realisierung einer Benutzerschnittstelle, können (in der Form von Applets) auf dem Browser direkt ausgeführt werden.

Application-Client Der Application-Client läuft als eigenständiges Java-Programm in einer eigenen Java-Maschine und kommuniziert über RMI (Remote Message Invocation) direkt mit dem Middle-Tier. Dabei hat er die Möglichkeit, wie der Internet-Browser, mit dem Web-Container zu kommunizieren. Zusätzlich hat ein Application-Client aber auch direkten Zugriff auf den EJB-Container und damit auch auf die darin ablaufenden EJBs.

Java-Web-Start-enabled-rich-Client Dies ist ein Client der ähnlich wie der Application-Client als eigenständiges Java-Programm läuft. Er basiert auf JFC/Swing und nutzt zur Kommunikation mit der J2EE Plattform XML und HTTP(S) Techniken. Er soll durch den Einsatz von Techniken wie „Java-Webstart“ [Mic03d] und JNLP [Mic03f] eine bessere Integration J2EE basierter Programme in den Anwender-Desktop ermöglichen.

Wireless Client Hierunter werden drahtlos angebundene Clients, die in der J2ME-Umgebung laufen, verstanden.

Da im Meta-Akad-Projekt nur die ersten beiden der genannten Client-Typen zum Einsatz kommen, sei für weitergehende Informationen über die anderen Client-Typen auf die Literatur verwiesen. [SSJtET02]

Die mittlere Schicht, Middle- oder Business-Tier genannt, wird durch einen Application-Server realisiert. Dieser stellt den Web-Container und den EJB-Container zur Verfügung. Zusätzlich stellt ein solcher Application-Server noch eine ganze Reihe von Diensten zur Verfügung:

- Namensdienst (JNDI)
- Transaktionsverwaltung (JTA)
- Nachrichtendienst (JMS)
- Datenbankanbindung (JDBC)
- E-Mail Unterstützung (JavaMail)

Eine komplette Auflistung aller von der J2EE Spezifikation geforderten Dienste und eine Beschreibung ihrer Funktionsweise findet sich in [Mic01]. Der Applikations-Server bietet meist noch weitere, über die Forderungen der J2EE Spezifikation hinausgehende, Dienste an. So bietet der im Meta-Akad-Projekt eingesetzte Orion-Server[1] zusätzlich eine

[1] Weitere Informationen zum Orion-Application-Server sind unter *www.orionserver.com* zu finden.

dynamische Lastverteilung.

Die unterste Schicht in Abbildung 5.1 stellt die Enterprise-Information-Tier (EIS-Tier) dar. Diese Schicht bietet Methoden und Funktionalitäten zur Datenspeicherung und zur Datenhaltung. Dabei handelt es sich in der Regel um ein Datenbanksystem mit einer entsprechenden Java-Anbindung (JDBC). Der Application-Server, der die JDBC Verbindungen verwaltet, bietet in der Regel zur Kommunikation mit dem DBVS „Pooling-Technologien“ an. Unter einem Pool von DB-Verbindungen ist eine Menge von offenen Verbindungen zu verstehen, die von den J2EE Komponenten gemeinsam genutzt werden können. Fordert eine solche Komponente eine DB-Verbindung beim Application-Server an, so übergibt dieser der Komponente ein entsprechendes Verbindungs-Objekt. Dieses wird allerdings erst dann an eine „echte“ DB-Verbindung gebunden, während die Komponente dieses Objekt verwendet. In der übrigen Zeit kann die echte DB-Verbindung von anderen Verbindungs-Objekte verwendet werden. Für den Entwickler auf J2EE Seite ist dieses Verhalten völlig transparent, ihm erscheinen die J2EE Verbindungs-Objekte als reale DB-Verbindungen. Diese Technik erlaubt es auf J2EE Seite wesentlich mehr Verbindungen „parallel“ halten zu können als es auf Seite des DBVS zugelassen wird.

5.2 J2EE Komponenten und der Applikations-Server

Die für J2EE entwickelten Komponenten laufen in den Containern eines J2EE konformen Applikations-Servers. Wie schon erwähnt, unterscheidet J2EE zwei Container-Typen: Den Web-Container und den EJB-Container. Diese werden im Folgenden eingehende beschrieben. Daran anschließend, wird die Komposition der einzelnen J2EE Komponenten zu einem Gesamtsystem erläutert.

5.2.1 Web-Container

Der Web-Container stellt die Laufzeitumgebung für die so genannten Web-Komponenten dar. Web-Komponenten sind Komponenten, die über einer HTTP-Verbindung mit einem Web-Client (Internet-Browser) kommunizieren. Die J2EE Spezifikation nennt zwei unterschiedliche Komponenten-Typen: Java Servlets und Java Server Pages (JSP). Beide dienen im Grunde als Brücke für die Kommunikation zwischen einem Web-Client und einem EJB-Container. Ihre Hauptaufgabe liegt in der Verarbeitung von Anfragen (der Web-Clients) und der Umsetzung von Informationen in eine vom Client verwertbare Form. Der Web-Container unterstützt diese Komponenten dabei durch Bereitstellung von verschiedenen Diensten. So sieht die J2EE Spezifikation beispielsweise ein automatisches Session-Management durch den Web-Container vor. Dieses Session-Management erlaubt es auf einfache Weise unterschiedliche Web-Clients identifizieren zu können. Dadurch wird es möglich, für jeden Web-Client einen so genannten „Conversational-State“ zu verwalten. Dieser erlaubt es, eine „Historie“ der einzelnen Client-Aktionen anzulegen. Ohne diese

Funktionalität wäre eine serverseitige Unterstützung von Anwendungen, wie etwa der „Einkaufskorb“ eines Online-Shops, nicht möglich.

5.2.2 EJB-Container

Ein EJB-Container ist die Laufzeitumgebung der Enterprise-Java-Beans (EJB). Er erzeugt und verwaltet die einzelnen EJBs. Zusätzlich stellt er den EJBs eine Reihe von Diensten zur Verfügung (siehe Abschnitt 5.1).

Der EJB-Container unterscheidet zwischen drei Grundtypen von Komponenten bzw. Enterprise-Java-Beans: Entity-, Session- und Message-Driven-EJBs. Die ersten beiden EJB-Typen müssen, um in einem EJB-Container lauffähig sein zu können, zwei bestimmte Schnittstellen definieren und implementieren. Diese beiden Schnittstellen werden Remote- und Home-Interface genannt. Das Home-Interface beschreibt Methoden, die der Container benötigt, um Instanzen des jeweiligen EJBs erzeugen und verwalten zu können. Im Falle der Entity-EJBs werden im Home-Interface zusätzlich noch Methoden zum Auffinden (anhand bestimmter Kriterien) von einzelnen Instanzen zur Verfügung gestellt. Das Remote-Interface beschreibt alle Methoden die ein EJB zur Verfügung stellen möchte. Komponenten, die ein EJB nutzen möchten, können nur auf die im Remote-Interface beschriebenen Methoden zugreifen.

Entity-EJBs

Entity-EJBs dienen in erster Linie dazu, Daten des persistenten Speichers zur Verfügung zu stellen, bzw. dort zu speichern. Ein Entity-EJB stellt die Objekt-Ansicht der Daten des persistenten Speichers dar. Ein Entity-EJB stellt Methoden bereit, um Daten zu finden, zu lesen und zu schreiben. (Es sollte normalerweise keine Methoden um Daten zu verarbeiten besitzen. Dies ist Aufgabe der Session-EJBs.) Mittels einer eigenen Anfragesprache, der EJB-QL, wird dem Entity-EJB mitgeteilt welche Daten eine bestimmte Methode liefern soll. Der EJB-Container setzt eine solche EJB-QL Anfrage dann in eine entsprechende für das Enterprise-Information-System (EIS) verständliche Anfrage um (z.B. SQL). Ihre Persistenzverwaltung können die Entity-EJBs entweder selbst realisieren (Beanmanaged) oder dies dem EJB-Container überlassen (CMP - Container-Managed-Persistence). In [SSJtET02] werden Entity-EJBs in aller Ausführlichkeit beschrieben. Da ihr Einsatz im Meta-Akad-Projekt keine ausschlaggebende Bedeutung für die dort untersuchten Skalierungskonzepte hat, werden sie hier nicht ausführlicher behandelt.

Session-EJBs

Session-EJBs sind in J2EE die Komponenten, die die Geschäfts- oder Programmlogik enthalten. Aus Sicht des Application-Server sind Session-EJBs nicht persistent, das heißt: Sie werden nicht explizit im persistenten Speicher gesichert. Es gibt keine Vorkehrungen, um ihr Überleben bei einem System-Neustart zu sichern. Ihre maximale Lebenszeit ist somit auf die Laufzeit des Application-Server beschränkt. Der EJB-Container kennt zwei

Typen von Session-EJBs: Die Stateful-Session-EJBs und die Stateless-Session-EJBs.

Stateless-Session-EJBs haben keine clientspezifischen Zustände oder Datenfelder. Aus Sicht des Clients hat ein solches EJB also keinen Zustand. Alle Instanzen eines Stateless-Session-EJBs sind identisch, dadurch muss einem Client kein explizites EJB zugewiesen werden. Stateless-Session-EJBs sind somit die J2EE-Entsprechung der in Kapitel 4.4.2 beschriebenen zustandslosen Komponenten. Ihr Haupteinsatzgebiet ist demzufolge dort, wo eine große Zahl von Clients zu erwarten ist.

Ein zweites Einsatzgebiet für die Verwendung von Stateless-Session-EJBs, ist die Realisierung von wiederverwendbaren Hilfskomponenten. Ein Beispiel hierfür ist das DBAccess-EJB des Resultset-Prozessors (Seite 46). Dieses EJB dient lediglich dazu, SQL Anfragen, die an das verwendete Datenbanksystem angepasst sind, zu erzeugen. Die Realisierung als Stateless-Session-EJB erlaubt es dem EJB-Container, dieses EJB einer großen Anzahl von Clients (hier der RSPController) zur Verfügung zu stellen.

Im Gegensatz dazu sind Stateful-Session-EJBs über ihre gesamte Lebenszeit fest an einen Client gebunden. Sie entsprechen den zustandsbehafteten Komponenten aus Kapitel 4.4.3. Stateful-Session-EJBs können clientspezifische Daten (auch einen „Conversational-State“) direkt im EJB speichern.

Man erhält so beispielsweise ein großes Einsparungspotential bezüglich der Anfragen an das Datenbanksystem. Ein solches EJB kann etwa einen Datenbank-Cursor halten und mit diesem auf einer Tabelle navigieren, auch über mehrere Methodenaufrufe hinweg. Ein Stateless-Session-EJB müsste sich zu Beginn und am Ende jeder Methodenabarbeitung die Cursorposition aus einem persistenten Speicher holen bzw. sie dort abspeichern. Außerdem müsste das Stateless-Session-EJB eine Möglichkeit besitzen, mit der es verschiedene Clients unterscheiden kann, etwa eine eindeutige Identifikations-Nummer als Methodenparameter.

Message-Driven-EJBs

Mit der Version 2.0 der J2EE Spezifikation wurden Message-Driven-EJBs als dritter EJB-Typ eingeführt. Sie kommunizieren mit ihren Clients über den Java-Message-Service, dadurch erhalten sie die Möglichkeit, asynchron arbeiten zu können. Message-Driven-EJBs verhalten sich ähnlich wie Stateless-Session-EJBs, sie sind allerdings noch einfacher aufgebaut. Sie benötigen weder ein Remote- noch ein Home-Interface. Die einzige Methode die sie implementieren müssen heißt *onMessage*. Sie wird immer dann aufgerufen, wenn eine Nachricht für das EJB eingetroffen ist. Die gesamte weitere Verarbeitung der Nachrichten übernimmt dann diese Methode.

5.2.3 Komposition von Komponenten

Die Komposition der einzelnen Komponenten zu einem größeren Software-Projekt, wird durch die Deployment-Deskriptoren festgelegt. Ein solcher Deployment-Deskriptor ist eine

Abbildung 5.2: Die verschiedenen Deployment-Deskriptoren von J2EE.

XML-Datei, die das Zusammenspiel der einzelnen Komponenten beschreibt. Die Abbildung 5.2 stellt die verschiedenen Deployment-Deskriptoren innerhalb einer J2EE Applikation dar. Für jeden der Container existiert ein eigener Deployment-Deskriptor (ejb-jar.xml, web.xml). Zusätzlich existiert ein Deployment-Deskriptor für die gesamte Applikation (application.xml) und jeweils einer für jeden Application-Client[2] (application-client.xml).

Deployment-Deskriptoren enthalten zwei Arten von Informationen: [SSJtET02]

- Strukturelle Informationen. Hierunter ist die Beschreibung der unterschiedlichen Komponenten bezüglich ihrer internen und externen Abhängigkeiten, der Belegung ihrer Umgebungsvariablen und der von ihnen benötigten Ressourcen zu verstehen.
- Informationen zur Komposition der Komponenten. Dies sind Informationen darüber wie die einzelnen Komponenten zu einer gemeinsamen J2EE Applikation zusammengesetzt werden sollen.

Änderungen an den Strukturellen Informationen sollten vom „Anwender“ einer Komponente nicht vorgenommen werden. Um diese korrekt vornehmen zu können, benötigt man in der Regel Zugriff auf den Quell-Code der Komponente bzw. ein genaues Wissen über ihre internen Abläufe. Diese Informationen sollten in der Regel vom Entwickler der Komponente bereitgestellt werden.

Die Informationen zur Komposition der Komponenten hingegen sollten vom „Anwender“ einer Komponente festgelegt werden.

Durch die Deployment-Deskriptoren wird es möglich, einzelne Komponenten auszutauschen, ohne den Quell-Code der Komponenten ändern und neu übersetzen zu müssen. Beim Austausch einer Komponente ist lediglich darauf zu achten, dass sie die gleichen Home- und Remote-Interfaces implementiert wie die Komponente die sie ersetzt.

Im folgenden Kapitel wird das MetaAkad Projekt und die dort eingesetzten Techniken erläutert.

[2]Hierbei sind unterschiedliche Implementierungen von Application-Clients gemeint, nicht unterschiedliche Instanzen.

Kapitel 6

Meta-Akad Projekt

In diesem Kapitel wird das Meta-Akad Projekt beschrieben. Dabei werden hauptsächlich die bezüglich der Datenhaltung und -verarbeitung relevanten Techniken und Konzepte betrachtet. Zunächst soll ein kurzer Überblick über das Gesamtprojekt gegeben werden. Ausführliche Informationen zum Meta-Akad Projekt finden sich in [Fle02] und [Gei00].

6.1 Meta-Akad Projektübersicht

Das Meta-Akad Forschungsprojekt hat zum Ziel, ein System zu schaffen, dass den effizienten Einsatz elektronischer Lehr- und Lernmaterialien im akademischen Umfeld ermöglicht. Es soll einen möglichst schnellen, bedarfsgerechten Zugriff auf lokal oder im Internet gespeicherte Dokumente ermöglichen. Hierzu sollen geeignete technische Mittel entwickelt bzw. bereitgestellt werden. Desweiteren werden im Rahmen des Projektes die nötigen Organisationsstrukturen aufgebaut. Insgesamt umfasst das Projekt das Sammeln von Online-Lehr- und Lernmaterial, dessen Erschließung (Klassifizierung und Beschlagwortung), die inhaltliche und didaktische Bewertung der einzelnen Dokumente und das Bereitstellen dieser Informationen unter einer einheitlichen Nutzeroberfläche. Das Erschließen dieser Dokumente geschieht in einem (semi-)automatischen Verfahren. Die inhaltliche und didaktische Bewertung der Dokumente wird über eine spezielle Benutzerschnittstelle, dem Redaktionssystem, ermöglicht. Die zusätzlich zu einem Dokument zu verwaltenden Daten, also die Klassifizierung, die Beschlagwortung und die Bewertung, werden in diesem Zusammenhang auch als Meta-Daten bezeichnet.

Aus technischer Sicht war für das Meta-Akad Projekt ein System zur Verwaltung von Dokumenten/Links und den zugehörigen Meta-Daten zu erstellen. Als zusätzliche Anforderung sollte das System die Möglichkeit bieten, die referenzielle Integrität (Konsistenz, Aktualität und Gültigkeit) von internen und externen Dokumenten zu verwalten.

Es wurde beschlossen, dass Meta-Akad System auf Basis eines kommerziellen Datenbankverwaltungssystems unter Verwendung von Java-Technologie zu entwickeln. Die Nutzerschnittstelle, sowie das Redaktionssystem sollen mittels eines WWW-Browsers über das Internet zugänglich sein. Da zu erwarten ist, dass die Nutzerschnittstelle einer wesentlich

höheren Last ausgesetzt sein wird als das Redaktionssystem und umgekehrt das Redaktionsystem wesentlich mehr Funktionalität aufweisen soll als die Nutzerschnittstelle, sollen die beiden Anwendungsschnittstellen bereits auf den unteren Ebenen des Meta-Akad Systems unterstützt werden.

Zusammenfassend sind als nicht funktionale Anforderungen an das Meta-Akad System folgende Punkte zu nennen:

1. Flexible Dokumentenspeicherung.
2. Flexible Speicherung der Meta-Daten.
3. Integritätsverwaltung (für Dokumente und Meta-Daten).
4. Flexible, gut skalierbare Web-Schnittstelle.
5. Mächtige Redaktionsschnittstelle.
6. Schnittstelle zum automatischen Erschließen.

Zusätzlich zu diesen wurde noch eine Reihe von Anforderungen genannt die vom Java-Teil des Meta-Akad Systems implementiert werden sollten. Diese zu implementierenden funktionalen Anforderungen an die Software des Meta-Akad Systems waren:

1. Ausführen von einfachen und komplexen Suchanfragen.
2. Ausführen von Meta-Queries auf der Menge der Suchergebnisse:
   - Durchführen unterschiedlicher Sortierungen der Treffermenge.
   - Abfrage der Trefferanzahl.
   - Bestimmen der häufigsten Kategorien in der Treffermenge.
   - Bestimmen der häufigsten Schlagwörter in der Treffermenge.
   - Erzeugen und Abrufen von Übersichtsseiten der Treffermenge.
3. Baumartige Darstellung der Kategorien mit „Treffern“.
4. Erzeugen einer Schlagwortvorschlagliste entsprechend der Suchanfrage.

6.2 Entwurfskonzepte

Im Folgenden werden die Konzepte erläutert die im Rahmen der Meta-Akad Projektes entwickelt wurden um die zuvor genannten technischen Anforderungen erfüllen zu können.

Abbildung 6.1: Meta-Akad Funktionsschema.

6.2.1 DBVS

Als Basis wurde, wie bereits erwähnt, ein kommerzielles Datenbankverwaltungssystem (Informix 9.3 UC2) gewählt. Dieses erlaubt die effiziente Speicherung und Verwaltung der Dokumente und ihrer Meta-Daten. Auch der Zugriff auf diese Daten, besonders bei der Suche nach bestimmten Dokumenten anhand von Meta-Daten, wird durch ein DBVS bestens unterstützt. Zusätzlich bietet ein Datenbankverwaltungssystem weitreichende Möglichkeiten zur Integritätsverwaltung der Dokumente bzw. ihrer Meta-Daten.

6.2.2 Java 2 Enterprise Edition als Basis

Um die Nutzerschnittstelle und das Redaktionsystem realisieren zu können und dabei die geforderte Flexibilität und Skalierbarkeit zu garantieren, wurde das Meta-Akad System als verteiltes, mehrschichtiges, komponentenbasiertes System entworfen. Als Entwurfs und Laufzeitumgebung wurde J2EE gewählt. Diese bietet, wie bereits in Kapitel 5 beschrieben, eine breite Infrastruktur für den komponentenbasierten, mehrschichtigen und verteilten Systementwurf. J2EE erlaubt eine einfache Anbindung des Datenbankverwaltungssystems über seine JDBC-Schnittstelle. Auch die Web-Integration (über den Web-Container - siehe Kapitel 5.2.1) wird von J2EE sehr gut unterstützt.

6.2.3 Komponentenansatz

Dem Komponenten-Ansatz folgend, wurde das Meta-Akad System auf mehrere Komponenten verteilt. Die dadurch entstehende Möglichkeit, einzelne Komponenten auszutauschen, ohne den restlichen Quellcode des System ändern und neu übersetzen zu müssen, erlaubt eine hohe Flexibilität des Gesamtsystems. Auch die Möglichkeiten der Skalierbarkeit des Meta-Akad Systems werden durch den komponentenbasierten Entwurf deutlich verbessert (horizontale oder diagonale Skalierung - siehe Kapitel 3.1.3).

Die Abbildung 6.1 stellt die einzelnen Komponenten des Meta-Akad Systems dar. Im Folgenden werden diese Komponenten und ihre Aufgaben im Meta-Akad System kurz beschrieben. Ausführliche Beschreibungen der Query-Engine und des Resultset-Prozessors folgen in Kapitel 7.

In der obersten Schicht (Web-Tier) befinden sich alle Komponenten, die für die Anbindung der unterschiedlichen Client-Applikationen zuständig sind. Zur Zeit existieren im Meta-Akad System fünf solcher Komponenten. In Abbildung 6.1 ist zusätzlich noch der Lastgenerator, der zur Evaluierung des Systems dienen soll, dargestellt.

WebSearch Diese Komponente stellt die zuvor beschriebene Nutzerschnittstelle dar. Sie erlaubt es über eine Eingabemaske Suchanfragen an das System zu stellen, deren Ergebnisse zu sichten und einzelne Dokumente zu laden.

WebMaintain Diese Komponente dient als Schnittstelle zur Verwaltung administrativer Daten. Beispiele hierfür sind: Die Benutzerverwaltung, Systemparameter des DBVS verwalten oder das Erstellen von Sicherungskopien des DBVS starten.

Indexing-Tool Das Indexing-Tool dient zum automatischen Erfassen (zum Beschlagworten, Klassifizieren und Indizieren) vom Lehr- und Lerndokumenten.

WebServices WebServices stellt eine Web-Schnittstelle zum Im- und Exportieren von Dokumenten des Meta-Akad Systems dar. Diese Schnittstelle wird beispielsweise von dem zu Anfang dieses Kapitels erwähnten (semi-)automatischen Tool zur Datenerfassung verwendet. Ein weiteres Anwendungsbeispiel für diese Schnittstelle, ist der Austausch von Dokumenten zwischen dem Meta-Akad System und anderen Bibliothekssystemen (etwa nach dem OAI-Standard[1]).

Editing-Tool Diese Komponente realisiert das Redaktionssystem. Sie bietet die gleichen Möglichkeiten, Anfragen, deren Ergebnisse und Dokumente zu verwalten, wie die WebSearch Komponente. Zusätzlich stellt diese Komponente Funktionen zum Bearbeiten vom Meta-Daten, sowie erweiterte Such- und Sortierroutinen zur Verfügung.

Lastgenerator Die letzte Komponente die in Abbildung 6.1 dargestellt ist, der Lastgenerator, ist kein eigentlicher Bestandteil des Meta-Akad Systems. Seine Darstellung in der Abbildung soll lediglich verdeutlichen, an welcher Stelle der Lastgenerator in das System eingreift, um seine Messungen durchführen zu können. Eine ausführliche Beschreibung des Lastgenerators folgt in Kapitel 8.2.

Die in Abbildung 6.1 EJB-Tier genannte Schicht enthält, wie in J2EE Systemen üblich, die gesamte Anwendungslogik des Meta-Akad Systems. Sie besteht aus drei Komponenten, die jeweils intern nochmals in kleinere Funktionsblöcke aufgeteilt sind. Für die Komponenten des Web-Tiers sind allerdings nur die folgenden drei Hauptkomponenten direkt ansprechbar:

Query-Engine Sie dient zum Verarbeiten von Suchanfragen. Sie nimmt eine Anfrage entgegen, erstellt einen Suchbaum und erzeugt die passenden Anfragen für das Datenbankverwaltungssystem und den Volltextindizierer (ASP-Seek). Die Ergebnisse dieser Anfragen führt die Query-Engine in einer gemeinsamen Ergebnistabelle zusammen und speichert diese in der Datenbank. Alle weiteren Operationen des Clients bezüglich dieser Anfrage werden auf dieser Ergebnistabelle ausgeführt.

[1] Open Archives Initiative, siehe *http://www.openarchives.org*

Resultset-Prozessor Dieser übernimmt die Verarbeitung der durch die Query-Engine erzeugten Ergebnistabellen. Der Resultset-Prozessor bietet hierzu Methoden zum Sortieren und Vorverarbeiten der XML-Daten. Zusätzlich erlaubt er es, die zuvor bereits erwähnten Meta-Queries auf den Ergebnistabellen auszuführen. Um die unterschiedlichen Anforderungen der Redaktions- und der Webschnittstelle optimal zu unterstützen, wurden zwei unterschiedliche Versionen der Resultset-Prozessors konzipiert. Sie sind jeweils für ihren Einsatzzweck optimiert und bieten teilweise sogar unterschiedliche Funktionalitäten (siehe Kapitel 7).

XML-Prozessor Der XML-Prozessor dient zum Management der XML-Dokumente. Er bietet Funktionen um die Meta-Daten der Dokumente auf XML-Attribute abzubilden bzw. die Attribute in Meta-Daten zu überführen. Zusätzlich bietet er Im- und Exportfunktionen für Meta-Akad Dokumente.

Die unterste Ausführungsschicht im Meta-Akad System, der EIS-Tier, beinhaltet zwei Komponenten zur Datenhaltung und -verarbeitung.

DBVS Das zentrale Datenbankverwaltungssystem übernimmt, wie schon eingangs erläutert, die Speicherung und Verwaltung der Dokumente und ihrer Meta-Daten.

ASP-Seek ASP-Seek ist ein spezialisierter Volltextindizierer. Er erlaubt es, effizient Suchanfragen über den gesamten Dokumenten-Text laufen zu lassen. So können die Nutzer des Systems, zusätzlich zur Suche in den Meta-Daten, direkt die Dokumente durchsuchen.

6.2.4 Beispiel einer einfachen Anfrage

Abbildung 6.2: Vereinfachte Darstellung des Ablaufes einer Anfrage an das Meta-Akad System.

Die Abbildung 6.2 stellt eine Übersicht der Abläufe bei der Abarbeitung einer Suchanfrage durch das Meta-Akad System. Nachdem ein Anwender in der entsprechenden Suchmaske der Web-Schnittstelle eine Suchanfrage eingegeben hat wird diese an der Query-Prozessor übermittelt. Dieser wertet die Anfrage aus und erzeugt entsprechende Anfragen für das DBVS und das ASP-Seek System. Die Ergebnisse dieser beiden Anfragen führt der Query-Prozessor in einer gemeinsamen Ergebnistabelle zusammen, welche er in der Datenbank speichert. Das erfolgreiche durchführen der Suchanfrage meldet der Query-Prozessor anschließend an die Web-Schnittstelle zurück. Dabei übergibt er dieser auch eine Referenz auf die erzeugte Ergebnistabelle. Die Web-Schnittstelle fordert daraufhin die erste Übersichtsseite mit den Suchergebnissen beim Resultset-Prozessor an. Dabei wird ebenfalls die Referenz auf die Ergebnistabelle übermittelt. Der Resultset-Prozessor sortiert die

Ergebnisse und lädt die benötigten Daten aus der Datenbank. Diese übermittelt er an die Web-Schnittstelle.

Eine ausführliche Beschreibung dieser Vorgänge ist im Kapitel 7.3 zu finden.

6.3 Probleme des Resultset-Prozessors

Eine der beiden bereits erwähnten Versionen des Resultset-Prozessors wurde als zustandslose Komponente (Stateless-Session-EJB) implementiert. Dadurch ergeben sich für diese Komponente (statelessRSPCtrlEJB) einige konzeptionelle Probleme die im Folgenden besprochen werden sollen.

6.3.1 Die Cursor Problematik

Ein Datenbank-Cursor erlaubt es der Applikation sequentiell über eine Tabelle bzw. das Ergebnis einer SQL-Anfrage zu iterieren. Da das statelessRSPCtrlEJB keine Zustände kennt, ist es ihm nicht möglich einen Datenbank-Cursor über mehrere Methodenaufrufe hinweg offen zu halten. Dadurch geht bei der Iteration über die Suchergebnisse am Ende der Bearbeitung einer Methode der DB-Cursor verloren. Die Funktionale Anforderung Übersichtsseiten der Ergebnismenge liefern zu können hat allerdings zur Folge, dass der DB-Cursor auf der (von der Query-Engine in der Datenbank erzeugten) Ergebnistabelle über mehrere Methodenaufrufe hinweg benötigt wird. Zur Iteration über diese Tabelle muss folglich eine Möglichkeit gefunden werden den DB-Cursor, oder wenigstens seine Position auf der Tabelle, außerhalb des statelessRSPCtrlEJB zu sichern.

Im folgenden Abschnitt wird das Stateless-Cursor-Konzept als Lösung für diese Problematik vorgestellt. Daran anschließend werden zwei weitere Lösungsmöglichkeiten vorgestellt und mit dem Stateless-Cursor-Konzept verglichen.

Stateless Cursor Konzept

Um die Funktionalität eines DB-Cursors erhalten zu können, wurde als Lösung das „Stateless-Cursor-Konzept“ entwickelt. Dieses erlaubt es während der Abarbeitung einer Methode einen regulären DB-Cursor zu verwenden. Am Ende der Abarbeitung einer Methode wird die Position des DB-Cursors in einem persistenten Speicher gesichert. Soll in der Folgezeit (während des Laufs einer anderen Methode) der DB-Cursor weiter verwendet werden, so wird ein neuer DB-Cursor erzeugt und die Position aus dem persistenten Speicher rekonstruiert. Auf diese Weise lässt sich die Position eines DB-Cursors bei zustandslosen Komponenten (bzw. ihren Instanzen) über die „Methodengrenze“ hinweg retten.

Abbildung 6.3: Das Sichern und Rekonstruieren eines DB-Cursors im Stateless Cursor Konzept.

Im Meta-Akad System wird zur Sicherung der Position des DB-Cursors ein zusätzliches Attribut in der Ergebnistabelle hinzugefügt. Dieses enthält eine fortlaufende Nummer. Wird die Ergebnistabelle nach dieser fortlaufenden Nummer sortiert, so entspricht beim Iterieren über diese Tabelle diese Nummer der Position des DB-Cursors. Soll der während eines Methodenlaufs verwendete DB-Cursor gesichert werden, so genügt es, die Nummer des Tupels an der aktuellen Position des DB-Cursors zu sichern. Zu diesem Zweck existiert im Meta-Akad System eine spezielle Verwaltungstabelle. Sie enthält unter anderem ein Feld für den Namen der Ergebnistabelle, sowie eines für die Position des Cursors (auf der Ergebnistabelle). Soll die Position des Cursors wieder rekonstruiert werden, so wird zunächst der entsprechende Eintrag aus der Verwaltungstabelle gelesen. Anschließend wird ein regulärer DB-Cursor erzeugt und entsprechend positioniert.

Ein Nachteil dieses Konzeptes ist der hohe Kommunikationsaufwand zum Sichern und Rekonstruieren der Cursor-Position. Inwieweit die erwarteten Vorteile (bei der Skalierbarkeit) einer zustandslosen Implementierung durch den so erhöhten Kommunikationsaufwand wieder ausgeglichen werden, soll die Evaluierung des Meta-Akad Systems zeigen (siehe Kapitel 9).

Ein weiterer Nachteil dieses Konzeptes ist die Einschränkung auf einen Stateless-Cursor pro Ergebnistabelle. Für jeden weiteren Cursor müsste ein weiteres Feld in der Verwaltungstabelle existieren. Zusätzlich müsste ein Mechanismus geschaffen werden, der es erlaubt, die richtige Cursor-Position aus den gespeicherten auszuwählen. Ansonsten bestünde die Gefahr eine falsche Cursor-Position zu rekonstruieren. Im Meta-Akad System wird das Stateless-Cursor-Konzept nur in Verbindung mit den Ergebnistabellen eingesetzt. Da diese immer explizit einem Client zugeordnet sind (und nur dieser darauf zugreift), genügt es, einen Cursor pro Ergebnistabelle verwalten zu können.

In Abschnitt 6.3.2 wird eine Erweiterung dieses Konzeptes erläutert, die es erlaubt, mehrere unterschiedliche Sortierungen für eine Ergebnistabelle effizient zu unterstützen.

In der Praxis existieren bereits verschiedene Techniken, die möglicherweise die hier beschriebene Funktionalität bieten könnten. In den beiden folgenden Abschnitten werden stellvertretend zwei alternative Lösungsmöglichkeiten vorgestellt und für die Zwecke des Meta-Akad Projektes bewertet.

Materialized-Query-Tables

IBM hat mit der Version 8 seiner Datenbank DB2-Systems die Materialized-Query-Tables (MQT) als Nachfolger der bis dahin eingesetzten Automated-Summary-Tables (AST) eingeführt (siehe [Nag02]). Bei den MQTs handelt es sich um Ergebnistabellen von Anfragen, die die DB2 entweder automatisch, oder explizit auf Benutzeranweisung hin erzeugt. Bei weiteren Anfragen prüft der Optimierer der DB2 ob es bereits eine MQT gibt die zur Beantwortung der Anfrage herangezogen werden kann. Ist dies der Fall, so schreibt der Optimierer die Anfrage so um, dass diese die vorhandenen MQTs ausnutzt. Das Umschreiben der Anfragen geschieht transparent für den Benutzer, so dass auf Anwendungsseite keinerlei Anpassungen notwendig sind. Aus dieser Technik ergeben sich erhebliche Vorteile

bei aufeinanderfolgenden, sich ähnelnden Anfragen.

Wie schon erwähnt, hat die DB2 zwei unterschiedliche Modi um MQTs zu erzeugen. Im automatischen Modus entscheidet der Optimierer aufgrund einer Analyse der Anfrage, ob das Ergebnis als MQT gespeichert werden soll. Im expliziten Modus kann der Benutzer das Anlegen einer MQT selbst bestimmen. In beiden Fällen kümmert sich die DB2 um die Konsistenzhaltung der MQTs. Der Benutzer hat allerdings die Möglichkeit, verschiedene Strategien (Immediate/Deferred) zu wählen (siehe [Nag02]).

Im Meta-Akad System könnte möglicherweise durch den Einsatz von MQTs auf das explizite Speichern der Ergebnismenge in der Datenbank verzichtet werden. Im Bereich der Meta-Queries sind hierbei keine Probleme zu erwarten. Man müsste jeweils die Meta-Query als Subquery in die ursprüngliche Anfrage einbetten. Der DB2 Optimierer sollte dies wiederum erkennen und die entsprechende MQT heranziehen. Die dabei zu erwartenden Geschwindigkeitsvorteile sind sogar größer, als beim im Meta-Akad System angewandten Konzept, da der Optimierer auch die MQTs anderer Anfragen heranzieht. Dies könnte sich besonders bei einer Folge von ähnlichen Anfragen stark positiv bemerkbar machen. Allerdings fehlt hier die explizite Tupelnummerierung und somit auch die Möglichkeit die Cursorposition in einer Verwaltungstabelle zu speichern. Auch die Möglichkeit mehrere verschiedene Sortierungen speichern zu können (siehe Abschnitt 6.3.2) ist hier nicht gegeben, da in die MQTs keine zusätzlichen Daten eingefügt werden können. Ein weiteres Problem bei diesem Ansatz stellen die Ergebnisse der Anfrage an den Volltextindizierer dar. Diese stehen nicht unter der Kontrolle des DBVS und können somit auch nicht über die MQTs abgedeckt werden.

Materialized-Views

Das Oracle Datenbanksystem bietet seit der Version 9i ein ähnliches Konzept: Die so genannten Materialized-Views (MViews). Diese verhalten sich in großen Teilen ähnlich wie die MQTs, es gibt allerdings auch einige signifikante Unterschiede. So müssen die MViews immer explizit erzeugt werden. Außerdem sollten zusätzlich Abhängigkeiten zwischen Attributen, auch über mehrere Tabellen hinweg, deklariert werden. Genaueres hierzu findet man in [Hob01].

Sind die MViews erzeugt, kann der Optimierer die Anfragen, genau wie bei der DB2, transparent umschreiben. Die Einsatzmöglichkeiten für MViews im Meta-Akad Projekt sind im wesentlichen die Gleichen, wie die der MQTs. Dies gilt allerdings auch für die den MQTs geltenden Einschränkungen bezüglich Speicherung von Cursor-Position, Sortierungen und Integration des Volltextindizierers.

Ein weiterer Nachteil dieser beiden Lösungen ist ihre System- bzw. Herstellerabhängigkeit, während das Stateless-Cursor-Konzept unabhängig von dem zugrundeliegenden Datenbankverwaltungssystem ist.

6.3.2 Sortierungen

Eine Anforderung an die zustandslose Variante des Resultset-Prozessors, war das effektive Durchführen von Sortierungen der Ergebnismenge nach vorgegebenen Attributen. Um eine einmal durchgeführte Sortierung nicht immer wieder wiederholen zu müssen (zustandslose Verarbeitung), wurde ein Konzept erstellt, das es erlaubt Sortierreihenfolgen zu speichern. Hierzu wurde das Stateless-Cursor-Konzept erweitert: Es wird nicht nur ein zusätzliches Attribut für die Cursor-Position eingeführt, sondern eine Reihe von Attributen. Jedes dieser Attribute enthält eine fortlaufende Nummerierung für die Tupel in der Ergebnistabelle. Diese fortlaufenden Nummernfolgen entsprechen jeweils einer bestimmten Sortierreihenfolge. In der Verwaltungstabelle wird, zusätzlich zur Cursor-Position, die aktuelle Sortiermethode gespeichert. So kann jederzeit ermittelt werden, nach welchen Attribut die Ergebnistabelle zu sortieren ist.

Um die Tupel in einer bestimmten Reihenfolge zu lesen, genügt es die Ergebnistabelle nach dem entsprechenden Attribut zu sortieren und mit einem DB-Cursor darüber zu iterieren. Muss die Cursorposition gesichert werden, so wird an Cursorposition die Nummer des Attributs nach dem sortiert wurde in die Verwaltungstabelle geschrieben. Zusätzlich wird noch der Name des Attributs nach dem sortiert wurde in der Verwaltungstabelle gespeichert.

Soll die Position des DB-Cursors (unter Beibehaltung der Sortierreihenfolge) rekonstruiert werden, so wird die Positionsnummer und der Name des entsprechenden Attributs aus der Verwaltungstabelle gelesen. Die Ergebnistabelle kann nun sortiert nach diesem Attribut gelesen werden. Ein neuer DB-Cursor kann an die richtige Position gesetzt und die ursprüngliche Iteration fortgesetzt werden.

Jedem Attribut in der Ergebnistabelle wird hierbei ein festes Sortiermuster zugeordnet. Derzeit werden vier unterschiedliche Sortiermuster in jeweils auf- und absteigender Reihenfolge unterstützt (siehe 7.2.2). Wird die Sortierung geändert, so sieht die Sortier-Komponente zunächst in der Ergebnistabelle nach. Findet sie dort in der entsprechenden Spalte bereits eine Sortierung vor, genügt, wie bereits erläutert, ein simples „ORDER BY“ um die Ergebnismenge wie gewünscht zu sortieren. Ist in der entsprechenden Spalte noch keine Sortierreihenfolge eingetragen, so wurde dieses Sortiermuster bisher noch nicht verwendet. Die Sortier-Komponente muss somit eine entsprechende Sortierung erst noch vornehmen. Hierzu werden alle Ergebnistupel referenziert und entsprechend der gewünschten Attribute (Sortiermuster) sortiert. In der Ergebnistabelle wird daraufhin in der, dem Sortiermuster entsprechenden Spalte die Sortierreihenfolge eingetragen. Auf diese Weise kann bei wiederholter Umsortierung das Referenzieren der Ergebnistupel eingespart werden. Unnötige Sortier-Durchläufe werden vermieden, da die Sortierfolge für jedes Sortiermuster erst bei ihrer erstmaligen Verwendung erzeugt wird. Das Ändern der Sortierfolge kann somit, bis auf den Erstfall, ebenfalls als eine Meta-Query gesehen werden.

Kapitel 7

Meta-Akad Realisierung

Im Folgenden werden der Query-Controller, die unterschiedlichen Varianten des Resultset-Prozessors sowie einige seiner Hilfskomponenten beschrieben. Den Abschluss dieses Kapitels bildet eine ausführliche Beschreibung der Abläufe im Meta-Akad System bei ausführen eine Suchanfrage.

7.1 Query-Engine

Die Query-Engine ist die Komponente im Meta-Akad System, die die Anfragen des Web-Tiers entgegennimmt und ausführt. Die Abbildung 7.1 stellt die Query-Engine des Meta-Akad-Systems detailliert dar. Dieser Abschnitt beschreibt die Abläufe innerhalb der Query-Engine bei Abarbeitung einer Suchanfrage:

Hat der Query-Controller (QECtrl) eine Anfrage erhalten, so führt er zunächst mittels eines Parsers eine Wortzerlegung dieses Suchstrings durch. Dabei erkennt der Parser auch zusätzliche Bedingungen/Attribute im Suchstring, wie etwa logische „und“- Verknüpfungen von Wörtern oder das gezielte Verwenden bestimmter Suchattribute (z.B.: RVK=„test“). Die so erhaltene Suchwortliste (inkl. der zusätzlichen Attribute) überführt der Parser in eine Baum-Darstellung, den Query-Tree (siehe [Web03]). Diesen Baum übergibt er dann an den Optimierer, der die Knoten im Query-Tree anhand einer Menge von heuristischen Optimierungsregeln neu anordnet. Dadurch soll eine möglichst kostengünstige (aus DBVS Sicht) Abarbeitung der Suchanfrage ermöglicht werden. Anschließend wird der optimierte Query-Tree an den Mediator weitergereicht. Dieser erzeugt jeweils eine Suchanfrage (in einer geeigneten Anfragesprache) für das DBVS und das ASP-Seek System. Die Antworten dieser beiden Systeme nimmt der Mediator entgegen und führt sie in einer gemeinsamen Ergebnistabelle zusammen (doppelte Einträge werden dabei eliminiert). Der Name dieser Ergebnistabelle bildet schließlich den Rückgabewert an den jeweils aufrufenden Client.

Abbildung 7.1: Darstellung der Query-Engine-Komponente des Meta-Akad-Systems.

7.2 Resultset-Prozessor

In diesem Abschnitt werden die verschiedenen Implementierungen des Resultset-Prozessors erläutert eingehen erläutert. Zunächst werden im folgenden Abschnitt die allgemeinen Eigenschaften und Funktionsweisen des RSP-Controllers erläutert. Daran anschließend werden die Besonderheiten der einzelnen Implementierungen des RSP-Controllers erläutert.

7.2.1 Der RSP-Controller

Der RSP-Controller ist die zentrale Komponente des Resultset-Prozessors. Er übernimmt die Verarbeitung der vom Query-Prozessor erzeugten Ergebnistabellen. Dabei soll er vor allem die in 6.1 beschriebenen Meta-Queries sowie Bereichsanfragen auf der Ergebnismenge unterstützen. Zusätzlich bietet er die Möglichkeit die aus der Datenbank gelesenen XML-Dokumente vor der Übergabe an den Web-Tier mittels eines XSLTC-Prozessors vorzuverarbeiten.

Der RSP-Controller ist als Enterprise-Java-Bean realisiert, das heißt: Jede Implementierung des RSP-Controllers muss das RSPCtrlHome- und das RSPCtrlRemote-Interface implementieren. Die Definition dieser beiden Interfaces ist in Anhang B zu finden. Da in der J2EE Umgebung keine zwei Komponenten zur gleichen Zeit die gleichen Home-Interfaces implementieren dürfen[1], wurde für die Home-Interfaces eine Klassenhierarchie angelegt. Die Abbildung 7.2 stellt diese Hierarchie dar. Für jede Implementierung des RSP-Controllers wurde ein eigenes Interface abgeleitet. Dadurch ist es möglich durch ändern des Deployment-Deskriptors die jeweils zu verwendende Implementierung des Resultset-Prozessors festzulegen. Der fastLaneReader stellt in diesem Zusammenhang allerdings eine Ausnahme dar, denn er ist nicht als Java-Enterprise-Bean implementiert. Genaueres zu den Gründen hierzu findet sich in Abschnitt 7.2.5.

Abbildung 7.2: Resultset-Prozessor: Home- und Remote-Interfaces

Zusammenfassend könnte man sagen: Die Query-Engine führt die „Selektion“ und der RSP-Controller die „Projektion“ der Suchergebnisse durch.

Zur Unterstützung des RSP-Controllers wurden eine Reihe von zusätzlichen Komponenten bzw. Klassen implementiert. Die folgenden Abschnitte beschreiben die von allen Implementierungen des RSP-Controllers gemeinsam genutzten Module:

DBAccess

Die DBAccess Komponente ist als Stateless-Session EJB realisiert. Sie stellt allerdings dennoch eine Besonderheit dar, denn sie implementiert nicht wie bei Session-EJBs üblich, ein

[1] Ist dies dennoch der Fall, so kann der Applikationsserver keine eindeutige Zuordnung zwischen Home-Interface und der entsprechenden Klasse vornehmen. Zum Beispiel wählt der Orion-Server automatisch immer die erste Klasse aus, die das gegebene Home-Interface implementiert.

Remote-Interface, sondern ein Local-Interface. Dieses Local-Interface hat die gleiche Funktion wie das Remote-Interface, lediglich die horizontale Skalierbarkeit ist eingeschränkt. Die Verwendung des Local-Interface hat zur Folge, dass eine Instanz dieser Komponente nur innerhalb des EJB-Containers erzeugt werden kann in dem die Instanz ihrer Client-Komponente läuft. Der Vorteil: Die Kommunikation zwischen einem Client und einer Instanz des DBAccess-EJB ist immer lokal. Dadurch kann der Applikationsserver auf RMI (Remote Message Invocation) Techniken verzichten und statt dessen bei der Parameterübergabe einfach Referenzen auf die zu übermittelnden Objekte übergeben.

Die eigentliche Aufgabe des DBAccess-EJB ist, eine zusätzliche Abstraktionsschicht zwischen dem EJB-Tier und dem DBVS (dem EIS-Tier) zu bilden. So muss beim Wechsel des DBVS lediglich das DBAccess-EJB angepasst werden. Dies wurde nötig, da im Resultset-Prozessor einige herstellerspezifische Funktionen des Informix DBVS verwendet wurden. Das in Abschnitt 7.2.2 beschriebene sortBean verwendet beispielsweise solche Funktionen.

Die Funktionsweise des DBAccess-EJB ist relativ einfach: Es hat für jede im Resultset-Prozessor benötigte SQL-Anweisung eine eigene Methode, die den entsprechenden SQL-String zurückliefert. So können die einzelnen SQL-Anweisungen auf eine sehr einfache Art und Weise an den jeweiligen Herstellerdialekt angepasst werden.

rspXMLHandler

Dieser stellt die Schnittstelle des Resultset-Prozessors zum Apache-XSLTC Prozessor dar. Der rspXMLHandler erlaubt es, XML Dokumente mittels eines XSL Stylesheets in die benötigte Form zu konvertieren. Dazu nimmt er als Parameter einen Text-String, der ein XML-Dokument enthält, sowie den Namen eines Stylesheets entgegen. Nach erfolgreicher Konvertierung liefert der rspXMLHandler das neue XML-Dokument wieder in Form eines Text-Strings als Rückgabewert.

Die Stylesheets werden bereits beim Übersetzen des Meta-Akad System vom XSLTC-Prozessor in Java-Klassen überführt. Durch diese Technik kann das wiederholte Parsen der Stylesheets vermieden und dadurch eine signifikante Leistungssteigerung erreicht werden.

rspHelper

Die rspHelper Klasse enthält alle Methoden, die in den unterschiedlichen Implementierungen des Resultset-Prozessors identisch sind. So kann besonders bei der Entwicklung viel überflüssige Arbeit eingespart werden.

Das im Folgenden statelessRSPCtrlEJB genannte EJB ist für den Einsatz im Zusammenspiel mit der Web-Schnittstelle (Web-Search Komponente) optimiert, das statefulRSPCtrlEJB für den mit der Redaktionsschnittstelle (Edit-Tool). Die beiden übrigen Implementierungen des RSPControllers (das cachingRSPCtrlEJB und der fastLaneReader) werden im Meta-Akad-System nicht eingesetzt. Sie wurden lediglich realisiert, um die ihnen zugrundeliegenden Konzepte testen zu können.

7.2.2 statelessRSPCtrlEJB

Das statelessRSPCtrlEJB ist als Stateless-Session-Bean realisiert. Es soll vor allem im Zusammenspiel mit der Web-Search Komponente des Web-Frontends zum Einsatz kommen. Da dieses Bean keine Zustände speichern kann, kommt hier das in Abschnitt 6.3.1 beschriebene Stateless-Cursor-Konzept voll zum Tragen. Die Methodenaufrufe haben aus diesem Grund alle das gleiche Muster:

1. Eine Methode wird aufgerufen und erhält als Parameter, neben den von der Methode benötigten, ein Resultset-Handle. Dieses Handle dient dazu, die zur aufrufenden Session gehörende Ergebnistabelle zu finden. Zunächst muss das Bean allerdings eine DB-Verbindung aufbauen.

2. Nun müssen die notwendigen Statusinformationen aus der Tabelle resultset aus der Datenbank gelesen werden. Dies sind Informationen wie etwa die aktuelle Cursorposition, die eingestellte Sortiermethode oder der Name der entsprechenden Ergebnistabelle.

3. Die Methode führt nun ihre eigentliche Aufgabe durch (zählen der Treffer in der Ergebnistabelle, laden der ersten X Einträge, etc.). Kommt es im Ablauf der Methode zu keinerlei Statusveränderungen, so kann die Methode sich an dieser Stelle beenden, bzw. ihren Rückgabewert liefern. Ansonsten muss aber noch der folgende Schritt durchlaufen werden.

4. In diesem letzten Schritt speichert die Methode geänderte Statusinformationen in der Tabelle resultset und schließt die Verbindung zur Datenbank.

Das Beispiel am Ende dieses Kapitels verdeutlicht die Abläufe innerhalb des statelessRSPCtrlEJB nochmals. Die fehlenden internen Zustände des statelessRSPCtrlEJB, zwingen es dazu, alle über die Dauer eines Methodenaufrufs hinweg benötigten Informationen in der Datenbank zwischenzuspeichern bzw. sie von dort zu lesen. Der dadurch entstehende Kommunikationsaufwand könnte sich bezüglich der Skalierbarkeit des Systems als Flaschenhals erweisen. Die beiden *stateful*-Varianten des RSP-Controllers versuchen genau diese Problematik zu umgehen in dem sie Informationen direkt in der jeweiligen Instanz zwischenspeichern.

sortBean

Das sortBean stellt, für die statelessRSPCtrlEJB Implementierung, die nötigen Funktionalitäten zur Sortierung der Ergebnistabelle zur Verfügung. Zur Zeit werden vom sortBean, wie bereits in 6.3.2 erwähnt, vier unterschiedliche Sortierungen in jeweils auf- und absteigender Reihenfolge unterstützt:

```
  create function sequence_func (init boolean default 'f')
       returning integer
     define global seq_func int default 0;
     if init = 't' then
          let seq_func = 0;
     else
          let seq_func = seq_func + 1;
     end if
     return seq_func;
end function;

--zur Ausführung:
execute function sequence_func('t') --initialisiert die Funktion
execute function sequence_func('f') --zählt hoch

--Anwendung in einer SQL-Anweisung:
select *,sequence_func('f') as order from resultset order by method into
table resultset_tmp
```

Beispiel 7.1: Eine Stored-Procedure für das Informix DBMS.

- match_asc, match_desc - Entspricht der Rangfolge des Volltextindizierers.
- general_asc, general_desc - Breitensuche: Entsprechend der Klassifizierung als allgemein eingestufte Dokumente werden zuerst geliefert.
- special_asc, special_desc - Tiefensuche: Als speziell eingestufte Dokumente zuerst anzeigen.
- new_asc, new_desc - Sortiert nach Einstelldatum der Dokumente in das System.

Die Sortierung der Ergebnistabellen soll möglichst direkt durch das DBVS erfolgen, die zu sortierenden Tupel sollen dabei nicht an das sortBean übertragen werden. Da die Sortierreihenfolge in der Ergebnistabelle vermerkt werden soll, ergibt sich hier ein großes Problem: Das Informix DBVS bietet keine Funktion, die sortierten Tupel automatisch durchzuzählen und somit auch keine Möglichkeit, die Sortierreihenfolge in der Tabelle festzuhalten.

Zur Lösung dieses Problems erzeugt das sortBean für jeden Sortierlauf eine Stored-Procedure wie sie Beispiel 7.1 zeigt. Nachdem diese Funktion initialisiert wurde, lässt sich mit ihrer Hilfe eine aufsteigende Folge von Zahlen erzeugen. Die unter dem Stichwort „Anwendung“ in Beispiel 7.1 genannte SQL Anweisung, sortiert die Tabelle *resultset* anhand des Attributs *method* und schreibt die Ergebnisse in die Tabelle *resultset_tmp*. In dieser neuen Tabelle enthält das Attribut *order* für jedes Tupel seine Position in der zuvor erzeugten Sortierreihenfolge.

Der Ablauf eines Sortiervorgangs durch das sortBean ist wie folgt:

1. Erzeugen der Stored-Procedure.

2. Initialisieren der Stored-Procedure.

3. Sortieren unter Verwendung der Stored-Procedure. Ergebnis in eine temporäre Tabelle schreiben.

4. Die Stored-Procedure löschen.

5. Den Inhalt der Ausgangstabelle löschen.

6. Die Daten aus der temporären Tabelle in die Ausgangstabelle kopieren.

7. Die temporäre Tabelle löschen.

Das sortBean nutzt zur Erzeugung der hierzu nötigen SQL Anweisungen das zuvor beschriebene DBAccess-EJB. Dieses liefert in diesem Fall eine Liste von SQL-Strings zurück, die durch das sortBean der Reihe nach abgearbeitet werden. Dadurch ist es möglich, zu dem Sortieralgorithmus Anweisungen hinzuzufügen bzw. daraus zu entfernen, ohne das sortBean selber verändern zu müssen. Soll der Sortieralgorithmus geändert werden, muss lediglich das DBAccess-EJB angepasst werden.

7.2.3 statefulRSPCtrlEJB

Das statefulRSPCtrlEJB ist, wie der Name es schon sagt, als Stateful-Session-Bean realisiert. Dadurch entsteht eine feste Bindung des Beans an die zuständige Web-Session, bzw. den zugehörigen Client. Da das RSPCtrlRemote-Interface dies so vorsieht, wird auch hier bei jedem Methodenaufruf das Resultset-Handle mitgeführt. Benötigt wird es allerdings nur beim ersten Aufruf einer beliebigen Methode. Zu diesem Zeitpunkt wird mittels des Resultset-Handle der Name der Ergebnistabelle festgestellt und zur weiteren Verwendung gespeichert. Der zweite Schritt des zuvor beschriebenen Beans muss also lediglich ein Mal durchgeführt werden. Alle anfallenden Statusinformationen können direkt im Bean zwischengespeichert werden, womit auch der vierte Schritt entfallen kann.

Um ein ständiges Auf- und Abbauen der Verbindung zur Datenbank zu vermeiden, hält das statefulRSPCtrlEJB über seine gesamte Lebenszeit ein DB-Verbindung offen. Aus diesem Grund kann auf das Stateless-Cursor-Konzept verzichtet werden und ein echte DB-Cursor zur Navigation auf der Ergebnistabelle verwendet werden.

Treten die statefulRSPCtrlEJBs in großer Zahl auf, so könnten die DB-Verbindungen bzw. die DB-Cursor zu einer knappen Ressource werden. Um die Zahl der offenen DB-Verbindungen bzw. der DB-Cursor zu reduzieren und trotzdem die Vorteile des statefulRSPCtrlEJB zu erhalten, wurde das cachingRSPCtrlEJB entwickelt.

7.2.4 cachingRSPCtrlEJB

Das CachingRSPCtrlEJB ist eine Abwandlung des statefulRSPCtrlEJB. Es verhält sich genau wie das statefulRSPCtrlEJB, lediglich die Methoden zum Abrufen von XML-Dokumenten werden durch einen Cache unterstützt. Zu diesem Zweck wurden die beiden Klassen RSPCache und fetchThread implementiert, sie stellen die benötigte Funktionalität zur Verfügung.
Die Objekte der Klasse RSPCache dienen zum Zwischenspeichern (Cachen) der XML-Dokumente. Sie stellen alle benötigten Methoden zum Zwischenspeichern und Verwalten der XML-Dokumente zur Verfügung.

Die Klasse fetchThread implementiert das Runnable-Interface von Java 2 (siehe [Fla00] und [WR00]). Dadurch wird es möglich Instanzen der fetchThread Klasse als eigene Java-Threads zu starten. Diese Threads laufen parallel zur erzeugenden Instanz und können so Arbeiten „im Hintergrund“ erledigen. Die Aufgaben des fetchThread sind das Lesen von XML-Dokumenten aus der Datenbank und deren Speicherung im Cache.

Abbildung 7.3: Ablauf der getFirstResultSetEntriesAsXML Methode im CachingRSPCtrlEJB.

Die Abbildung 7.3 zeigt ein Sequenzdiagramm das den Ablauf der getFirstResultSetEntriesAsXML Methode beschreibt. Nach dem Aufruf der Methode, prüft die Instanz des CachingRSPCtrlEJB zunächst, ob die angeforderten Dokumente bereits in ihrem Cache (RSPCache-Objekt) vorhanden sind. Ist dies nicht der Fall (siehe Abbildung 7.3), so ließt das CachingRSPCtrlEJB die ersten X (= die Anzahl der angeforderten Dokumente) Tupel aus der Datenbank und speichert diese im Cache. Anschließend erzeugt und startet die Instanz des CachingRSPCtrlEJB einen neuen fetchThread und übergibt ihm als Parameter eine Referenz auf das Cache Objekt. Da der fetchThread unabhängig von der CachingRSPCtrl Instanz läuft, kann diese nun die angeforderten Dokumente an den Client zurückliefern. Der fetchThread beginnt parallel dazu mit dem Lesen von zusätzlichen Tupeln, die er ebenfalls im Cache speichert. Er führt dies fort bis die dreifache Menge der ursprünglich angeforderten Daten im Cache gespeichert ist.

Nachdem sich der fetchThread beendet hat, stehen somit die ersten 3*X XML-Dokumente im Cache bereit. Da die Clients, im Besonderen der WebClient, die Zahl der angeforderten Dokumente in der Regel nicht variieren, können die nächsten beiden getNextResultSetEntriesAsXML-Anfragen direkt aus dem Cache beantwortet werden (siehe Abbildung 7.4). Erst danach müssen weitere Tupel aus der Datenbank geladen werden.

Abbildung 7.4: Ablauf der getNextResultSetEntriesAsXML Methode im CachingRSPCtrlEJB.

Insgesamt ergibt sich durch den Einsatz des RSPCache und des fetchThread für den Client eine minimal höhere Wartezeit beim Aufruf der getFirstResultSetEntriesAsXML

Methode. Im Gegenzug werden die ersten beiden getNextResultSetEntriesAsXML Methodenaufrufe sehr schnell beantwortet. Ob im praktischen Einsatz diese Implementierung Vorteile gegenüber dem statefulRSPCtrlEJB zeigen kann, hängt stark vom Verhalten der jeweiligen Nutzer ab. Ließt eine große Zahl der Nutzer nur die erste Übersichtsseite, so ist kein nennenswerter Vorteil für das CachingRSPCtrlEJB zu erwarten. Durch die längere Laufzeit der getFirstResultSetEntriesAsXML Methode kann sogar im Extremfall die Gesamtleistung sinken.

Die redundante Speicherung der XML-Dokumente auf der EJB-Ebene bringt aber auch einige Nachteile mit sich. So ist zu erwarten, dass sich der Speicherbedarf auf der EJB-Ebene beträchtlich erhöht. Ein weiteres Problem stellt die Konsistenz der Daten im Cache dar: Der Client sieht Änderungen in der Datenbank erst wenn er eine neue Anfrage stellt oder die Sortierung wechselt. Die im Meta-Akad System gespeicherten Daten ändern sich nur selten, so dass auf Konsistenzsichernde Maßnahmen an dieser Stelle verzichtet werden kann.

Ein weiterer Nachteil dieser Implementierung ist die Belastung des Datenbankverwaltungssystems: Diese erhöht sich in jedem Fall, da für jeden Client mindestens die für die ersten drei Übersichtsseiten benötigten Tupel gelesen werden.

Ob die Vorteile dieser Implementierung ihre Nachteile aufwiegen können, sollen die in Kapitel 9 beschriebenen Messungen zeigen.

Abschließend ist anzumerken, dass diese Implementierung eine Besonderheit des Orion-Servers ausnutzt, denn dieser erlaubt es die fetchThreads als eigenständige Java-Threads zu starten. Die J2EE Spezifikation verbietet eigentlich das explizite Starten von Java-Threads durch eine Applikation. Um ein asynchrones Verhalten einer Komponente zu realisieren sieht die J2EE Spezifikation den Einsatz von Message-Driven-EJBs vor. Der Orion-Server in der Version 1.5.4 wie er im Meta-Akad Projekt eingesetzt wird unterstützt diese jedoch noch nicht, so dass auf die Lösung mit den Java-Threads zurückgegriffen wurde.

7.2.5 fastLaneReader

Abbildung 7.5: *Fast-Lane-Reader* Design Pattern von Sun Microsystems.(Entnommen aus [Mic03c])

Die fastLaneReader Komponente nimmt, wie bereits mehrfach angedeutet, eine Sonderstellung ein: Sie ist nicht als Enterprise-Java-Bean realisiert, womit sie im Meta-Akad System nicht einsetzbar ist, ohne größere Änderungen vornehmen zu müssen. Sie wird dementsprechend auch nur zu Evaluierungszwecken vom Lastgenerator verwendet. Dieser setzt eine speziell hierfür angepasste Version der userThreads ein (siehe Kapitel 8.2.5). Der fastLaneReader bietet die gleiche Funktionalität wie das statefulRSPCtrlEJB.

Diese Implementierung folgt dem *Fast-Lane-Reader* Design Pattern der *Java BluePrints* Sammlung von Sun Microsystems (siehe [Mic03c]). Die Abbildung 7.5 zeigt die von Sun vorgeschlagene Struktur des *Fast-Lane-Reader* Patterns. Ein Client wie in der Abbildung

dargestellt, existiert in dieser Implementierung nicht. Die userThreads des Lastgenerators stellen direkt die *Application-Component* dar. In einer angepassten Version des Meta-Akad Systems würde an dieser Stelle die Web-Search Komponente eingesetzt werden.

Das Pattern sieht vor, dass die *Application-Component* direkt, ohne den Umweg über den EJB-Container, mit dem *Fast-Lane-Reader* auf die Datenbank zugreift. Dadurch gehen natürlich alle Dienste die der EJB-Container bereitstellt verloren, z.B.: Die Transaktionsverwaltung oder das Connection-Pooling. Aus diesem Grund soll der *Fast-Lane-Reader* auch nur zum schnellen, sequentiellen lesen von Daten eingesetzt werden. Schreibende Zugriffe aus dem *Fast-Lane-Reader* auf die Datenbank sollten möglichst vermieden werden.

Um Schreibzugriffe zu vermeiden, speichert die fastLaneReader Implementierung alle Statusinformationen direkt in Instanzvariablen, Sortierreihenfolgen berechnet sie (genau wie das StatefulRSPCtrlEJB) nicht im Voraus. Da die fastLaneReader Instanzen im Kontext des Lastgenerators laufen, werden sie auch in dessen Java-VM ausgeführt. Somit ergibt sich ein entscheidender Unterschied zu den EJB-Implementierungen, die im Kontext des Applikationsservers laufen. Das hat zur Folge, das der kritische Punkt (Flaschenhals) unter hoher Last auf Seite der Clients (Lastgenerator) zu suchen ist. Dadurch kann kein direkter Vergleich zwischen dem fastLaneReader und den übrigen Implementierungen vorgenommen werden. Die Messungen mit der fastLaneReader Implementierung sollen lediglich die Größenordnungen der Skalierbarkeit erfassen.

Abbildung 7.6: Der *Fast-Lane-Reader* beim Abrufen der ersten Übersichtsseite.

Die Abbildung 7.6 stellt die Abläufe im Meta-Akad System bei Verwendung der fastLaneReader Implementierung durch den Lastgenerator. Wie man sieht umgeht der fastLaneReader den Resultset-Prozessor vollständig, die gesamte Programmlogik des Resultset-Prozessors ist im fastLaneReader (und somit im Lastgenerator) enthalten.

7.3 Ausführliches Beispiel einer Anfrage

Abbildung 7.7: Sequenzdiagramm der Abarbeitung einer Suchanfrage. Teil 1 Query-Engine.

Im Folgenden wird der Ablauf einer Suchanfrage eines Web-Clients ausführlich dargestellt. In diesem Beispiel wird von einem „normalen“ Anwender ausgegangen, d.h. dass sein Client (ein Web-Browser) über eine HTTP-Verbindung mit einer Instanz der Web-Search Komponente kommuniziert. Dies hat wiederum zur Folge, dass die „stateless“ Variante des Resultset-Prozessors zum Einsatz kommt. Einige der Abläufe im Meta-Akad-System sind in diesem Beispiel vereinfacht wiedergegeben, da hier nur der prinzipielle Ablauf einer Suchanfrage erläutert werden soll. Die Abbildung 7.7 stellt den ersten Teil (Query-Engine) des Ablaufes der Suchanfrage in Form eines Sequenzdiagrammes dar. Abbildung 7.7 zeigt den zweiten Teil (Resultset-Prozessor) des Ablaufes.

Erhält die WebSearch Komponente bzw. eine ihrer Instanzen eine Suchanfrage in Form einer Zeichenkette, so startet dies eine neue Instanz der Query-Engine und übergibt ihr diese Zeichenkette. (Siehe Punkt ① in der Abbildung.)

Die Query-Engine führt nun mittels ihres Parsers eine Wortzerlegung der Zeichenkette durch und erzeugt einen dazu analogen Suchbaum (Query-Tree). Anschließend ordnet der Optimierer der Query-Engine den Suchbaum um. Dadurch soll eine, bezüglich ihrer geschätzten Laufzeitkosten, optimale Repräsentation des Baumes erzeugt werden. Im folgenden Schritt erzeugt der Mediator zum Suchbaum äquivalente Anfragen für das DBMS und das ASP-Seek System. Nachdem die Query-Engine die Anfragen an diese beiden Systeme übermittelt hat, wartet sie auf die Ergebnisse dieser Anfragen. Diese Ergebnisse führt dann der Mediator, unter Eliminierung von Duplikaten, in einer gemeinsamen Ergebnistabelle zusammen. Der Name dieser Tabelle und einer ihr zugeordnete Identifikationsnummer (rsID in der Abbildung) werden in einer Verwaltungstabelle gespeichert. Zur weiteren Verarbeitung der Suchergebnisse, muss künftig nur noch die rsID der Ergebnismenge mitgeführt werden. Entsprechend bildet die rsID den einzigen Rückgabewert der Query-Engine an die Web-Search Instanz (Punkt ②).

Abbildung 7.8: Sequenzdiagramm der Abarbeitung einer Suchanfrage. Teil 2 Resultset-Prozessor: setSortMethod

Die Web-Search Instanz startet im Anschluss daran den Resultset-Prozessor (Punkt ③ in Abbildung 7.8) und ruft in diesem die „setSortMethod“ Methode mit den Parametern rsID (zur Identifikation der Ergebnistabelle) und method (Name der gewünschten Sortiermethode) auf. Da es sich in diesem Beispiel um die zustandslose Implementierung des Resultset-Prozessors handelt und diese keine Historie der Clientaktionen kennt, muss der Resultset-Prozessor zunächst alle evtl. benötigten Verwaltungsinformationen (z.B.: Tabellenname, Cursorposition, aktuell zu verwendende Sortiermethode, bereits durchgeführte Sortierungen) aus der Datenbank lesen, bevor er den Sortiervorgang starten kann. Dieser besteht aus einer Folge von SQL-Anweisungen, so dass die Sortierung direkt im DBVS erfolgen kann. Das zur Zeit im Meta-Akad-System eingesetzte DBVS (Informix) unterstützt nicht alle hierzu benötigten Funktionen. Der Resultset-Prozessor erzeugt deswegen eine „Stored-Procedure“, die die fehlende Funktionalität (durchnummerieren der Tupel) nachbildet. Nach der erfolgreichen Sortierung speichert der Resultset-Prozessor die geänderten Verwaltungsinformationen (durchgeführte Sortierung, aktuell zu verwendende Sortierreihenfolge) in der Datenbank ab. Anschließend meldet er an die WebSearch Instanz ein „OK“ zurück (Punkt ④).

Abbildung 7.9: Sequenzdiagramm der Abarbeitung einer Suchanfrage. Teil 3 Resultset-Prozessor: getFirstAsXML

Die Abbildung 7.9 zeigt den dritten und letzten Teil des beispielhaften Ablaufs: Das Abrufen der ersten X Dokumente durch die WebSearch Instanz. Diese ruft hierzu die „getFirstAsXML“ Methode mit den Parametern rsID und X (Die Anzahl der zu liefernden

Tupel) im Resultset-Prozessor auf (Punkt ⑤). Dieser muss wieder alle benötigten Statusinformationen aus der Datenbank lesen, da er die vorangegangenen Aktionen „vergessen“ hat. Nun kann der Resultset-Prozessor die ersten X Tupel aus der Datenbank lesen. Da die Sortierreihenfolge schon im vorherigen Schritt berechnet wurde, genügt ihm hier ein „ORDER BY“ Zusatz auf das entsprechende Attribut in der Ergebnistabelle. Anschließend verarbeitet der Resultset-Prozessor die gelesenen XML-Daten mit einem (durch WebSearch konfigurierbaren) XSLT-Prozessor. Die so den Wünschen der Web-Search Instanz angepassten XML-Daten liefert der Resultset-Prozessor an die Web-Search Instanz zurück (Punkt ⑥). Zuvor muss er allerdings alle geänderten Statusinformationen in der Datenbank sichern.

Als letzten Schritt in der Abarbeitung der Suchanfrage liefert WebSearch die erhaltenen Daten zur Darstellung an den Web-Client (Punkt ⑦).

Dieses Beispiel zeigt deutlich die Nachteile der zustandslosen Implementierung: Hoher Kommunikationsaufwand, der „Conversational-State“ muss durch die Web-Search Instanz verwaltet werden (mitführen der rsID).

Die Vorteile sind hier nicht so deutlich sichtbar: Die Instanz des Resultset-Prozessors an Punkt ③ und an Punkt ⑤ muss nicht die gleiche sein. Nach den Punkten ④ und besonders ⑥, kann die Instanz des Resultset-Prozessors von anderen verwendet werden.

Kapitel 8

Quantitative und qualitative Evaluierung - Konzepte

In diesem Kapitel wird die quantitative und qualitative Evaluierung der verschiedenen zuvor vorgestellten Konzepte beschrieben. Zunächst wird das konzeptionelle Vorgehen und die daraus resultierende Implementierung beschrieben. Anschließend werden die einzelnen Messungen beschrieben und ihre Ergebnisse interpretiert.

8.1 Messverfahren

Im Folgenden werden die zur Evaluierung der verschiedenen Implementierungen des RSP-Controllers angewandten Konzepte beschrieben.

8.1.1 Konzepte

Um eine quantitative Bewertung der verschiedenen Implementierungen vornehmen zu können, wurde ein speziell an das Meta-Akad Projekt angepasster Lastgenerator entwickelt. Dieser Lastgenerator simuliert, in Anlehnung an die Ergebnisse von [Geo02], eine Gruppe von Benutzern die auf das System zugreifen. Das Messverfahren ist an die TPC-W Benchmark-Spezifikation [Cou99] und die darin beschriebenen Verfahren (siehe [Cou99, S. 81]) angelehnt. Um die Messungen vorzunehmen, werden unterschiedliche Aktionen durch den Lastgenerator (siehe 8.2) ausgeführt und dabei die Antwortzeiten gemessen. Folgende Aspekte verdienen hierbei besondere Beachtung:

Simulierte-Benutzer Der Lastgenerator startet und verwaltet die Simulierten-Benutzer (im Folgenden auch SB genannt), die eigenständig mit dem Meta-Akad System in Verbindung treten. Diese führen Aktionen durch, messen deren Laufzeiten und protokollieren diese. Jeder Simulierte-Benutzer soll in einer möglichst realistischen Art und Weise das Verhalten eines realen Benutzers nachahmen.

Web-Interaktionen Da die SB möglichst realistische Szenarien durchspielen sollen, wurden Web-Interaktionen eingeführt. Eine Web-Interaktion simuliert das reale Verhalten der Web-Schnittstelle auf EJB-Ebene. Eine Web-Interaktion stellt ein aus Sicht eines Anwenders atomares Ereignis dar. Sie entspricht einer Aktion die ein realer Anwender der Webschnittstelle durchführt: Beispielweise ist das Umsortieren der Ergebnismenge für den Nutzer der Web-Schnittstelle ein atomares Ereignis, da er lediglich den Knopf mit der Aufschrift „neu Sortieren“ drückt. Für das Meta-Akad System ergibt sich allerdings eine Folge von Aktionen die durchzuführen sind: Sortieren der Ergebnisse, rücksetzten des Offsets, erzeugen einer neuen Übersichtsseite. Innerhalb einer Web-Interaktion werden die gleichen Methoden in der gleichen Reihenfolge aufgerufen, wie es auch die reale Web-Schnittstelle bei der entsprechenden Aktion eines Anwenders machen würde. Die SB führen ausschließlich Web-Interaktionen durch um so die Aktionen eines realen Benutzers zu simulieren. Gemessen wird jeweils die Zeit vom Start der Web-Interaktion, also dem Senden der ersten Anfrage, bis zu dem Zeitpunkt, an dem alle zur Interaktion gehörenden Daten vollständig an den Client übertragen sind.

Session Ein Folge von Web-Interaktionen beginnend mit der *getFirstPage* Interaktion (siehe Seite 66) und endend mit der *userDone* Interaktion stellt eine Session dar. Eine Session ist das Äquivalent einer Web-Session im realen System. Jeder SB kann beliebig viele Sessions in einer Folge durchführen.

Think-Time Zwischen den verschiedenen Web-Interaktionen wartet der Simulierte-Benutzer eine bestimmte Zeitspanne, die Think-Time. So soll das Lesen der präsentierten Daten durch einen realen Benutzer simuliert werden. Um der unterschiedlichen Lesegeschwindigkeit realer Benutzer Rechnung zu tragen, wird hierfür jeweils ein Zufallswert innerhalb eines vorgegebenen Intervalls berechnet.

Durchsatz Um die Anzahl der Web-Interaktionen pro Zeitintervall messen zu können, wird die Systemzeit zu Beginn und am Ende jeder Interaktion gespeichert. In einem nachfolgenden Analyseschritt, kann aus diesen Daten die Summe aller in einem bestimmten Zeitintervall aktiven, bzw. abgeschlossenen Interaktionen bestimmt werden.

Fehler Tritt bei Abarbeitung einer Web-Interaktion ein Fehler auf, so wird dieser protokolliert um eine spätere Auswertung zu ermöglichen.

Ramp-up period Damit das System „warm laufen“ kann, wird die Messung erst nach einer vorgegebenen Zeitspanne begonnen.

Systemwerte Zusätzlich werden über die gesamte Testlaufzeit verschiedene Systemressourcen wie etwa die Hauptspeicher-, CPU-, Netz- oder IO-Auslastung beobachtet. Dadurch wird es möglich, Rückschlüsse auf Systemengpässe während des Messvorgangs zu ziehen.

Suchbegriff	#Treffer	Selektivität	Suchbegriff	#Treffer	Selektivität
standard	25	0.58%	near	277	6.47%
stochastik	25	0.58%	kontinuumsmechanik	282	6.59%
aller	26	0.61%	analysis	294	6.87%
effekt	26	0.61%	schwingung	295	6.89%
kombinatorik	26	0.61%	wissen	316	7.38%
komplexe	26	0.61%	methoden	326	7.61%
lernen	26	0.61%	algebra	333	7.78%
mechanism	26	0.61%	system	333	7.78%
newton	26	0.61%	wing	340	7.94%
second	26	0.61%	teil	342	7.99%
sein	26	0.61%	methode	343	8.01%
studenten	26	0.61%	introduction	355	8.29%
synthesis	26	0.61%	applet	360	8.41%
teaching	26	0.61%	stat	363	8.48%
various	26	0.61%	sammlung	367	8.57%
anatomy	27	0.63%	line	412	9.62%
approach	27	0.63%	intro	423	9.88%
biologie	27	0.63%	form	444	10.37%
endliche	27	0.63%	vorlesung	444	10.37%
kognitive	27	0.63%	lehre	473	11.05%
sehr	27	0.63%	lehrmaterial	479	11.19%
series	27	0.63%	didaktik	487	11.37%
wechselwirkung	27	0.63%	mechanik	491	11.47%
alberts	28	0.65%	material	538	12.56%
applications	28	0.65%	quanten	554	12.94%
atomphysik	28	0.65%	nach	595	13.90%
complex	28	0.65%	geschichte	613	14.32%
computational	28	0.65%	graph	657	15.34%
funktionentheorie	28	0.65%	allgemeine	1161	27.11%
graduate	28	0.65%	theorie	1191	27.81%
phasen	28	0.65%	mathematik	1305	30.48%
review	28	0.65%	mathe	1361	31.78%
sources	28	0.65%	math	1366	31.90%
denken	29	0.68%	physik	1431	33.42%

Tabelle 8.1: Auszug aus der Liste der Suchwörter mit Angabe ihrer Selektivität. Die Summe aller möglichen Treffer betrug zu diesem Zeitpunkt 4282.

Selektivität der Suchergebnisse Um fundierte Aussagen über die Laufzeit einer Suchanfrage treffen zu können, ist es nötig, die Anzahl der „Treffer“ dieser Suchanfrage zu kennen. Das Verhältnis der Treffer-Anzahl einer Anfrage zur Gesamtzahl der möglichen Treffer wird als die Selektivität einer Anfrage bezeichnet. Die in einem Messvorgang verwendeten Suchbegriffe werden aus einer Liste gewählt in der für jeden Begriff seine Selektivität bestimmt wurde. So kann man bei der Auswahl der Suchwörter gezielt Begriffe mit einer hohen bzw. niedrigen Selektivität auswählen. Die Tabelle 8.1 zeigt einen Ausschnitt aus dieser Liste.

8.1.2 Benutzer-Typen

Wie schon eingangs erwähnt, wurden statistische Daten über das „Anwenderverhalten im Umgang mit Suchmaschinen“ einer Studie der Firma iProspect [Geo02] entnommen. In dieser Studie werden die Daten einer E-Mail Umfrage aus dem Frühjahr 2002 statistisch ausgewertet und analysiert.

Für die Entwicklung des Lastgenerators von Interesse ist vor Allem die Antwort auf eine Frage, die von den Teilnehmern zu beantworten war:

„*When you are presented with search engine results, before you decide on one to click on, how many entries do you typically review?*“

Die möglichen Antworten auf diese Frage können der Tabelle 8.2 entnommen werden. Die Gruppen die auf die erst bzw. zweite Frage mit „Ja“ geantwortet haben, werden zu einem Benutzertyp zusammengefasst. Für den Lastgenerator ist diese Zusammenfassung nicht von Bedeutung, da die Benutzer aus beiden Gruppen in jedem Fall die erste Seite laden müssen.

Benutzer-Typ	mögliche Antworten	prozentualer Anteil
1	Only a few.	16.10
	The first page.	31.90
2	The first two pages.	23.00
3	The first three pages.	10.30
4	More than three pages.	8.70
5	The whole list.	9.90

Tabelle 8.2: *iProspect, Search Engine Branding Survey*: Umfrageergebnisse zum Nutzerverhalten im Umgang mit Suchmaschinen.

Die simulierten Benutzer werden analog der Umfrageergebnisse auf die fünf Benutzertypen verteilt. So ließt ein Benutzer von Typ eins nur die erste Seite der Suchergebnisse,

Abbildung 8.1: Der Ablauf des Lastgenerators und der Benutzer-Simulatoren.

bevor er sich für ein Ergebnis aus dieser Liste entscheidet oder seine Suche abbricht. Entsprechendes gilt für die Benutzer der Typen zwei und drei. Benutzer vom Typ vier lesen eine zufällig bestimmte Anzahl von Seiten im Bereich zwischen vier und der maximalen Seitenzahl. Die vom Typ fünf lesen alle Seiten der Ergebnisliste. Um bei sehr großen Ergebnismengen den Lauf eines Simulierten-Benutzers in einem überschaubaren Zeitrahmen zu halten, ist für die Typen vier und fünf allerdings die maximale Anzahl der Ergebnis-Seiten auf 20 beschränkt. Weitere möglicherweise bestehende Ergebnis-Seiten werden nicht abgerufen.

Im anschließenden Absatz wird die Implementierung des Lastgenerators beschrieben. Dabei wird jeweils herausgestellt, wie die zuvor aufgeführten Aspekte die Implementierung beeinflusst haben.

8.2 Der Lastgenerator

In diesem Abschnitt wird die Funktionsweise des Lastgenerators und der prinzipielle Ablauf eines Messvorgangs beschrieben. Zunächst werden allerdings kurz einige technische Implementierungsdetails erläutert.

8.2.1 Lastgenerator als Application-Client

Um die Entwicklung des Lastgenerators zu vereinfachen, wurde ein so genannter Application-Client, der „Rspclient“, implementiert. In der J2EE Umgebung versteht man hierunter einen Client, der über den J2EE-Applikationsserver verbreitet wird. Er läuft in einer eigenen Java-VM und greift direkt auf die EJBs des Application-Server zu. Hierdurch kann der HTTP-Zugriff auf die Web-Search Komponente und das damit verbundene Analysieren und Auswerten der zurückgelieferten HTML- und JSP-Seiten vermieden werden. Der Rspclient simuliert die Zugriffe der Web-Search Komponente auf der EJB Ebene. Somit umgeht der Rspclient vollständig den Web-Tier des Meta-Akad Systems. Da die Komponenten des Web-Tier allerdings bei den zu testenden Implementierungen immer identisch sind und die Messungen sich lediglich auf die EJB-Ebene beziehen, sollte durch ihre Umgehung keine Verfälschung der Messergebnisse auftreten.

8.2.2 Ablauf des Lastgenerators

Die Abbildung 8.1 zeigt den Ablauf des Lastgenerators und den eines userThreads. Zunächst liest der Lastgenerator die Konfiguration für den Messlauf aus einer XML-Datei ein. Für jeden zu simulierenden Benutzer erzeugt der Lastgenerator ein eigenes Logging-Objekt. Anschließend legt der Lastgenerator für jeden zu simulierenden Benutzer fest welchen der fünf Benutzertypen er simulieren soll. Als nächstes erzeugt der Lastgenerator für

jeden simulierten Benutzer einen Java-Thread (den userThread) und übergibt diesem die benötigten Parameter: Das Logging-Objekt, den auszuführenden Suchstring, eine eindeutige Kennnummer sowie Informationen über die Dauer des Messlaufs, die das Think-Time Intervall.

Der eigentliche Messvorgang wird dann durch Aufruf der Run-Methode in den einzelnen userThreads gestartet. Der Lastgenerator wartet bis alle userThreads beendet sind und gibt anschließend die in den Logging-Objekten gespeicherten Messergebnisse aus (bzw. speichert sie in einer Datei).

Um einzelne Messläufe durchzuführen hat der Lastgenerator einen interaktiven Modus, der es außerdem gestattet sämtliche Funktionen der zur Verfügung stehenden EJBs einzeln aufzurufen (zu debugging Zwecken). Um eine automatische Durchführung der Messläufe zu ermöglichen, besitzt der Lastgenerator zusätzlich noch einen „nicht interaktiven“ Betriebsmodus. In diesem nimmt er auf der Kommandozeile als Parameter den Namen einer XML-Konfigurationsdatei entgegen. Diese Datei enthält alle weiteren Informationen für den durchzuführenden Messlauf.

Im Folgenden werden nun, wie schon zu Beginn des Abschnittes angekündigt, die einzelnen Teil-Komponenten des Lastgenerators beschrieben.

8.2.3 Konfiguration des Lastgenerators

Um eine möglichst große Flexibilität des Lastgenerators zu erreichen, wurden die gesamten Parameter eines Messvorgangs in eine XML-Datei ausgelagert. Eine solche XML-Datei könnte dabei wie in Beispiel 8.1 aussehen.

Das Attribut beanType des Benchmark-Tags legt fest, welche Implementierung des Resultset-Prozessors auf der EJB-Ebene zu verwenden ist. Das User-Tag legt fest, wie viele Benutzer simuliert werden sollen und wie lange (in Minuten) die Simulation des Benutzers andauern soll. Hat ein Benutzer-Thread alle seine Interaktionen durchgeführt und die runTimeMin ist noch nicht abgelaufen, so beginnt er seinen Durchlauf von vorne. Im Page-Tag wird festgelegt, wie viel Prozent der Benutzer bis zu dieser Seite lesen. So bedeutet number=“1“ und readProbability=“60“, dass 60% der Benutzer nur die erste Seite lesen, number=“2“, readProbability=“20“ bedeutet 20% lesen genau die ersten beiden Seiten. Aus diesen Informationen wird die Verteilung der Benutzer auf die fünf Benutzertypen berechnet. Mit dem SearchString-Tag können beliebig viele Suchbegriffe angegeben werden. Diese werden sequentiell den simulierten Benutzern zugewiesen. Sind weniger Suchbegriffe in der Konfigurationsdatei, als Benutzer simuliert werden sollen, so werden die Suchbegriffe zufällig aus dieser Menge ausgewählt.

8.2.4 Logging-Komponente

Die Logger Komponente ist eine Java-Klasse, die beliebig viele Ereignisse mit Start- und Endzeit speichern kann. Zusätzlich besteht noch die Möglichkeit eine Anmerkung als Text-

```
  <?xml version=''1.0'' standalone=''yes''?>

<rsp:benchmark xmlns:rsp=''rspclient-config/'' beantype=''stateless''
name=''1_stateless''>
     <rsp:user numberofusers=''1'' runTimeMin=''5''/>
     <rsp:usertype number=''1'' probability=''50''/>
     <rsp:usertype number=''2'' probability=''20''/>
     <rsp:usertype number=''3'' probability=''10''/>
     <rsp:usertype number=''4'' probability=''10''/>
     <rsp:usertype number=''5'' probability=''10''/>
     <rsp:pagesize size=''10''/>
     <!-- Zeitangaben jeweils in ms. -->
     <rsp:thinktime upperboundary=''45000'' lowerboundary=''15000''/>
     <rsp:startdelay upperboundary=''45000'' lowerboundary=''15000''/>

     <!-- Ab hier folgen nur noch Suchbegriffe. Die Anzahl der zu
     erwartenden Treffer je Suchwort steht als Kommentar
     am Ende jeder Zeile.-->
<rsp:searchstring>configuration</rsp:searchstring><!-- 5 -->
<rsp:searchstring>atome</rsp:searchstring><!-- 11 -->
<rsp:searchstring>computer</rsp:searchstring><!-- 77 -->
<rsp:searchstring>physik</rsp:searchstring><!-- 1431 -->
</rsp:benchmark>
```

Beispiel 8.1: XML–Konfigurationsdatei für den Lastgenerator.

String zu jedem Ereignis zu speichern. Um die Daten unterschiedlicher Benutzer-Threads bei der Ausgabe besser unterscheiden zu können, markiert der Logger jedes Ereignis mit einer Kennung für den Thread.

Durch den Einsatz dieser Logging-Komponente können ein direktes Speichern der Log-Informationen und die damit möglicherweise verbundenen Verzögerungen vermieden werden. Ein weiterer Vorteil ist die zentrale Steuerung der Ausgabe der Log-Daten. Anpassungen im Ausgabeformat, wie sie für unterschiedliche Analysewerkzeuge nötig sind, lassen sich so leicht realisieren.

8.2.5 Simulierte Benutzer

Die eigentliche Simulation der Benutzer findet in den userThreads statt. Um die im Kapitel 8.1.1 beschriebenen Web-Interaktionen zu simulieren, existiert für jede Web-Interaktion eine eigene Methode im Benutzer-Simulator. Folgende Web-Interaktionen werden hierbei unterstützt:

getFirstPage Entspricht der Eingabe eines Suchstrings und dem Drücken des *Suchen*-Knopfes der Web-Seite. Sie beinhaltet also das Starten einer Anfrage, das Generieren der temporären Ergebnistabelle, das Setzten einer Sortiermethode, das entsprechende Sortieren und das Abrufen der ersten Übersichtsseite.

getNextPage Entspricht dem Drücken des *Nächste Seite*-Knopfes auf der Web-Seite. Sie ruft die nächste Übersichtsseite ab.

changeSortOrder Führt alle Aktionen zum Wechsel der Sortierreihenfolge durch: Sortieren, rücksetzen des Offsets und laden der ersten Übersichtsseite (entsprechend der neuen Sortierung). Diese Web-Interaktion entspricht somit dem Auswählen einer neuen Sortierung durch einen realen Benutzer.

getDocument Entspricht dem Auswählen eines Dokumentes aus einer Übersichtsseite.

userDone Entspricht dem Beenden der Web-Sitzung. Intern wird ein releaseResultset aufgerufen und die temporäre Ergebnistabelle wird gelöscht.

Nachdem ein Benutzer-Thread erzeugt wurde, wartet dieser zunächst, bis seine run-Methode aufgerufen wird (siehe Abbildung 8.1). Da bei der Simulation unter Umständen eine große Anzahl von Threads nahezu zeitgleich starten, kann es hier zu Problemen kommen. So würden beispielsweise alle userThreads ihre Suchanfrage in einem sehr kurzen Zeitintervall starten. Ein solches Szenario ist allerdings in einem realen Umfeld nicht zu erwarten. Um ein synchrones Ablaufen der Threads zu vermeiden, wartet jeder Thread nach Aufruf seiner Run-Methode eine bestimmte Zeitspanne. Die Dauer der Wartezeit wird in jedem Thread als Zufallswert bestimmt. Über die Konfigurationsdatei lässt sich allerdings ein Intervall angeben, in dem dieser Zufallswert liegen muss (siehe Seite 66).

Erst nach Ablauf dieser Zeitspanne startet der userThread die getFirstPage Web-Interaktion. Je nach Typ führt er im Folgenden ein bzw. mehrmals die getNextPage und changeSortOrder Interaktion durch. Den Abschluss bildet dann die userDone Interaktion um das Ende der Session zu markieren.

Direkt vor und nach jeder dieser Web-Interaktionen speichert der userThread die Systemzeit (in ms) in einer Variable. Bevor er die nächste Interaktion durchführt, ruft der Benutzer-Thread die logEvent Methode seines Logging-Objektes, mit den gemessenen Zeiten und dem Namen der Web-Interaktion als Parameter, auf. Dann wartet er noch eine, aus dem Think-Time Intervall, zufällig gewählte Zeitspanne um so das Lesen der gelieferten Daten durch den Benutzer zu simulieren.

Diesen Ablauf wiederholt der userThread entsprechend der in der Konfigurationsdatei Zeitspanne. Dadurch ist es möglich, das System vor der eigentlichen Messung erst „warm laufen“ zu lassen (Ramp-up-period). Näheres zu den einzelnen Messdurchläufen findet sich im folgenden Kapitel.

8.3 Auswertung der Messergebnisse

Mittels eines Unix Shell-Skriptes können die während eines Messlaufs angefallenen Log-Informationen analysiert und ausgewertet werden. Dazu schreibt der Lastgenerator am Ende des Messlaufs ein Log-Datei die alle während des Laufs Protokollierten Informationen in einem strukturierten Format enthält. Ein kurzer Auszug aus einer solchen Log-Datei zeigt Beispiel 8.2.

```
0:1:true:getFirstPageWebInteraction:1053027474610:1053027487053:12443:
0:1:true:getNextPageWebInteraction:1053027510900:1053027511047:147:
0:1:true:getNextPageWebInteraction:1053027530851:1053027539742:8891:
0:1:true:getNextPageWebInteraction:1053027562830:1053027563262:432:
0:1:true:doneReadingWebInteraction:1053028081410:1053028081501:91:
```

Beispiel 8.2: Ein kurzer Ausschnitt aus einer Logdatei eines Messlaufs.

Das Skript log2plot (siehe A.3) wertet diese Log-Dateien aus und erzeugt dabei eine Datei die die aggregierten Messergebnisse enthält. Das Format dieser Datei wurde so gewählt, dass eine direkte graphische Auswertung mit dem Programm gnuplot (siehe [Buß03] und [Cra03]) möglich ist. Die Datei mit den aggregierten Messergebnissen kann Daten mehrerer Messläufe enthalten. Die durch das Skript berechneten Daten sind:

- Anzahl der Clients (NoC).
- Summe der Antwortzeiten aller Interaktionen.(TfI).
- Gesamtzahl der Interaktionen des Messlaufs (NoI).
- Durchschnittliche Laufzeit einer Interaktion (ATI).
- Anzahl der Fehler während des Tests (E).
- Anzahl der Fehler pro Interaktion (EpI).
- Die Anzahl der Deadlock Fehler im Messlauf (D).
- Das Verhältnis von Deadlock Fehlern zur Anzahl der Interaktionen (DpI).

Kapitel 9

Quantitative und qualitative Evaluierung - Realisierung

Dieses Kapitel beschreibt die vorgenommenen Messungen und analysiert deren Ergebnisse aus.

9.1 Messumgebung

Bevor die Messläufe beschrieben werden, sollen an dieser Stelle zunächst die technischen Rahmenbedingungen erläutert werden. Um möglichst unverfälschte Messergebnisse zu erhalten, verwenden das Meta-Akad-System, der Lastgenerator und das Informix DBVS jeweils ein eigenes Rechnersystemen. Die Systeme sind miteinander über ein 100 MBit Netzwerk verbunden.

DBVS Das Informix DBVS in der Version 9.3 UC2 läuft auf einem Sun Enterprise Server unter dem Betriebsystem Solaris 8. Dieser Server verfügt über vier SPARC–2 CPUs mit jeweils 300MHz Taktfrequenz und 1.5 GB Hauptspeicher.

Application-Server Der Orion-Server läuft auf einem IBM e–Series Server mit vier Intel-Xeon Prozessoren (1.5 GHz) und 2 GB Hauptspeicher. Als Betriebsystem läuft darauf SUSE Linux in der Version 8.2.

Lastgenerator Der Lastgenerator und somit auch alle Simulierten-Benutzer laufen auf einem Personalcomputer mit AMD Athlon 1900+ CPU und 512 MB Hauptspeicher. Als Betriebssystem kommt hier Redhat Linux in der Version 8.0 zum Einsatz.

Während der Messläufe wurden Systemwerte, die möglicherweise Einfluss auf die Messergebnisse haben könnten, regelmäßig überprüft bzw. überwacht. Besonders die CPU-Belastung, die Hauptspeicherbelegung sowie der Datendurchsatz der Netzwerkschnittstellen waren dabei von großem Interesse. Dabei konnte beobachtet werden, dass zu keinem

Zeitpunkt, auf keinem der verwendeten Rechnersysteme die technische Leistungsfähigkeit der genannten Komponenten voll ausgeschöpft wurde. Somit kann ausgeschlossen werden, dass die Skalierbarkeit in den vorgenommenen Messläufen durch hardware-technische Gründe beschränkt wurde.

9.2 Messreihen - Erläuterung

Zur Evaluierung des Meta-Akad Systems und der vier Implementierungen des Resultset-Prozessors wurden etliche Messreihen durchgeführt. Eine Messreihe besteht dabei jeweils aus einer Folge von mehreren Messläufen mit steigender Anzahl der Simulierten-Benutzer. Innerhalb einer Messreihe werden außer der Benutzer- bzw. Clientanzahl keine weiteren Parameter verändert. Die Auswirkungen auf die Skalierbarkeit von Änderungen anderer Systemparameter wurden jeweils in eigenen Messreihen untersucht.

Da keine dedizierte Netzwerk-Infrastruktur zur Verfügung stand und außerdem das DBVS auch von anderen Projektteilnehmern genutzt werden sollte, wurden zur Reduzierung von äußeren Einflüssen, die Messreihen außerhalb der üblichen Bürozeiten durchgeführt (Nachts und am Wochenende). Teilweise wurde den anderen Nutzern des DBVS der Zugriff für die Dauer der Testläufe vollständig entzogen.

Zum Erzeugen der jeweils benötigten XML-Konfigurationsdateien einer Messreihe wurde ein Unix Shell-Skript (siehe Anhang A.2) entwickelt. Dieses Skript, genConfig genannt, erzeugt ausgehend von einer Basiskonfiguration, die für die einzelnen Messläufe einer Messreihe benötigten Konfigurationsdateien.

Die grafische Auswertung der Messergebnisse erfolgte mit dem Programm *gnuplot* (siehe [Buß03], [Cra03]). Seine Hauptaufgabe ist das Zeichnen von Funktions- und Messkurven aller Art. Es besitzt allerdings auch einige Funktionen zum Aufbereiten vom Rohdaten. Für die in diesem Kapitel dargestellten Messkurven wurde, um eine bessere Anschaulichkeit zu erreichen, eine Bezier-Funktion zur Glättung der Funktionsverläufe verwendet. Durch diese Funktion werden starke Ausreißer in der Menge der Messergebnisse relativiert. Das heißt es wird eine „Mittelwert-Kurve“ der Messergebnisse einer Messreihe erzeugt. Im Anhang C sind alle der im Folgenden dargestellten Messkurven nochmals in ihrem „Rohzustand“ abgebildet.

Zur Auswertung in dieser Diplomarbeit wurden die Ergebnisse von etwa 40 Messreihen herangezogen. Im Ablauf dieser Messreihen wurden ungefähr 550 einzelne Messläufe durchgeführt und dabei 425000 Suchanfragen abgearbeitet. Insgesamt wurden im Laufe dieser Messreihen schätzungsweise 1,25 Millionen Web-Interaktionen in der beschriebenen Messumgebung durchgeführt. Die Nettolaufzeit dieser Messungen betrug 300 Stunden.

9.3 Ergebnisse der Messungen

In diesem Teilkapitel werden die Ergebnisse der einzelnen Messreihen vorgestellt und bewertet. Zu jeder untersuchten Frage- oder Aufgabenstellung, wie beispielsweise „Bestimmen der generellen Leistungsfähigkeit der einzelnen Implementierungen.“ (siehe im Folgenden Abschnitt), wurden in der Regel jeweils vier Messreihen durchgeführt. Diese vier Messreihen sind jeweils, bis auf die verwendete Implementierung des Resultset-Prozessors, identisch. Im folgenden wird eine solche Vierergruppe von Messreihen als ein Test bzw. einen Testdurchlauf bezeichnet. Die Messergebnisse eines Testdurchlaufs werden jeweils in einer gemeinsamen Grafik zusammengefasst.

9.3.1 Generelle Leistungsfähigkeit

Um in einem ersten Testdurchlauf die generelle Leistungsfähigkeit des Meta-Akad Systems (in der beschriebenen Messumgebung) zu bestimmen, wurde für jede Implementierung des Resultset-Prozessors eine Messreihe durchgeführt. Hierbei wurden in der Messumgebung keinerlei künstliche Schranken (wie etwa eine Limitierung der maximalen Zahl von parallelen DB-Verbindungen) eingeführt.

Ein weiteres Ziel dieser Messreihen war es ein sinnvolles Intervall für die Zahl der zu simulierenden Benutzer der folgenden Messreihen zu bestimmen.

Abbildung 9.1: Die absolute Anzahl der durchgeführten Web-Interaktionen in Abhängigkeit von der Anzahl der simulierten Benutzer.

Die Abbildung 9.1 stellt die Gesamtzahl der in den jeweiligen Messläufen durchgeführten Interaktionen dar.

Erhofft wurde eine, mit der Anzahl der simulierten Clients, monoton steigende Zahl von durchgeführten Web-Interaktionen. Dies würde darauf hindeuten, dass das System bei dieser Last seine Leistungsgrenze noch nicht erreicht hat. Steigt die Anzahl der durchgeführten Web-Interaktionen, trotz wachsender Clientanzahl, nicht weiter an, so hat das System seine maximale Leistungsfähigkeit erreicht. Um ein weiteres Skalieren des Systems zu ermöglichen müsste analysiert werden, welcher Teil des Systems die Skalierbarkeit einschränkt und gegebenenfalls entsprechende Skalierungstechniken eingesetzt werden.

Abbildung 9.2: Die Fehlerrate pro Web-Interaktion.

Wie in der Abbildung 9.1 zu erkennen ist, steigt die Anzahl der durchgeführten Interaktionen für jede der vier Messreihen zunächst wie erhofft monoton an. Ab einer Zahl von 200 simulierten Clients sinkt allerdings die Anzahl der durchgeführten Web-Interaktion bei der StatelessRSPCtrlEJB- und der StatefulRSPCtrlEJB-Variante. Die beiden anderen Implementierungen gehen zunächst zu einer mehr oder weniger konstanten Anzahl

von durchgeführten Interaktionen über, bevor diese dann ebenfalls stark absinkt. Das erhoffte Verhalten konnte somit leider bei keiner der vier Implementierungen festgestellt werden. Das CachingRSPCtrlEJB und der FastLaneReader zeigen zwar bis zu einer Last von etwa 300 (bzw. 350) Clients ein gutes Skalierungsverhalten, brechen danach aber in ihrer Leistung stark ein. Das StatefulRSPCtrlEJB zeigt zwar bis zu einer Last von etwa 200 Clients ein genau so gutes Skalierungsverhalten wie die beiden zuvor Genannten. Wenn der Punkt seiner maximalen Leistungsfähigkeit überschritten wird, bricht es allerdings wesentlich schneller ein als der FastLaneReader bzw. das CachingRSPCtrlEJB. Das StatelessRSPCtrlEJB zeigt das schlechteste Skalierungsverhalten im Messfeld: Es zeigt von Anfang an weniger Leistung als die drei Anderen. Es skaliert deutlich schlechter (geringere Steigung der Messkurve) und bricht ebenfalls bei einer Last von 200 Clients ein.

In Abbildung 9.3 sind die durchgeführten Web-Interaktionen pro Sekunde dargestellt. Dieser Wert wird direkt aus der Absoluten Zahl der durchgeführten Interaktionen berechnet. Er spiegelt somit im Wesentlichen die gleichen Informationen wieder, wie die Abbildung 9.1. Man kann sehen, dass das StatelessRSPCtrlEJB und das StatefulRSPCtrlEJB ab einer Last von etwa 200 bis 250 Clients ein ungünstiges Skalierungsverhaltenzeigen. Der FastLaneReader und das CachingRSPCtrlEJB zeigen ab einer Last von 300 bis 350 Clients ebenfalls dieses Ungünstige Verhalten.

Wie Abbildung 9.2 illustriert, steigt die Fehlerrate im Gegensatz dazu mit zunehmender Last stark an. Dabei traten in allen Messreihen Fehler im Query-Prozessors auf. So hatte der Query-Prozessor häufiger Probleme beim Anlegen der Ergebnistabelle, das DBVS meldete die Fehlernummer -310 (Namenskonflikt beim Anlegen einer Tabelle). Eine Überprüfung des Algorithmus zum Erzeugen des Tabellennamen konnte keine Fehler aufdecken. Es konnte allerdings durch eine zusätzliche Messreihe eine mögliche Fehlerursache entdeckt werden. Die Namen der Ergebnistabellen werden in einer eigenen Tabelle (Resultset) gespeichert. Beim Generieren eines Namens für eine neue Ergebnistabelle sieht die Query-Engine in Resultset nach, ob der Name bereits vergeben ist. Ist dies nicht der Fall, so wird innerhalb einer Transaktion dieser Name in Resultset eingetragen und die entsprechende Tabelle angelegt. Problematisch könnte es also nur sein, wenn eine Ergebnistabelle existiert, deren Namen nicht in Resultset eingetragen ist. Um dies zu vermeiden, wird in der Methode releaseResultset das Löschen des Tupels mit dem Tabellennamen (aus Resultset) und das Löschen der entsprechenden Ergebnistabelle in einer gemeinsamen Transaktion durchgeführt. In der bereits erwähnten Messreihe zur Untersuchung dieses Verhaltens, wurde das Löschen des Tupels mit dem Tabellennamen aus Resultset aus den Programmcode entfernt. Wird also die releaseResultset Methode aufgerufen, so löscht diese die entsprechende Ergebnistabelle. Der Name dieser Ergebnistabelle bleibt allerdings weiterhin in Resultset eingetragen und somit für die weitere Verwendung gesperrt. Bei den durchgeführten Messläufen mit dem so modifizierten System, traten keine weiteren Fehler dieser Art auf. Es also zu vermuten, dass das DBVS trotz Transaktionsschutz das Löschen der Ergebnistabelle nicht wie erwartet durchführt. Ein Ausweg wäre möglicherweise den Isolations-Level der Datenbank zu erhöhen.

Ein weiteres Problem des Query-Prozessors waren besonders unter hoher Last häufig

auftretende Deadlock-Fehler. Der genaue Ursprung dieser Fehler konnte allerdings nicht ermittelt werden. Er ließ sich lediglich soweit eingrenzen, dass diese Fehler mit großer Wahrscheinlichkeit dem Orion-Server zugeschrieben werden können. Diese Deadlock-Fehler treten vor allem auf, wenn sehr häufig auf Entity-EJBs zugegriffen wird. Im Meta-Akad System wird allerdings fast ausschließlich lesend auf diese EJBs zugegriffen, so dass eigentlich keine Deadlock-Fehler auftreten dürften. Da diese Entity-EJBs automatisch durch den Orion-Server generiert werden, war eine genaue Ermittlung des Grundes für dieses Fehlverhalten nicht möglich. Die Auswirkungen dieser Deadlock-Fehler auf das übrige System sind allerdings sehr groß. Tritt diese Deadlock-Problematik auf, so sperrt der Orion-Server den Zugriff auf dieses Entity-Bean (und mitunter auch auf die Datenbank) für 90 Sekunden, um anschließend ein Rollback der entsprechenden Transaktion durchzuführen. In dieser Situation kommt es (während der Sperrzeit) bei jedem Simulierten-Benutzer der auf das gleiche Entity-Bean zugreifen will ebenfalls zu einem Deadlock-Fehler. Dies hat zur Folge, dass das Auftreten eines Deadlock-Fehlers eine ganze Reihe von weiteren Deadlock-Fehlern nach sich zieht.

Die speziellen Probleme bzw. Vorteile der verschiedenen Implementierungen des Resultset-Prozessors sind:

StatelessRSPCtrlEJB Da das StatelessRSPCtrlEJB die Datenbank nutzt um seine Zustandsinformationen dort zu sichern bzw. sie von dort zu rekonstruieren, ist hier die Belastung des Datenbanksystems wesentlich größer als bei den anderen Implementierungen. Durch den so entstandenen Mehraufwand bei der Abarbeitung jeder Web-Interaktion sinkt die Grundleistung und Skalierbarkeit des StatelessRSPCtrlEJB im Vergleich zu den übrigen Varianten des RSP-Controllers stark ab.

StatefulRSPCtrlEJB Das StatefulRSPCtrlEJB unterstützt im Gegensatz zur Stateless Variante eine wesentlich breitere Anzahl von Sortieroptionen. Um die Sortierung entsprechend einem gewünschten Attribut vornehmen zu können muss das StatefulRSPCtrlEJB die Abhängigkeiten zwischen den einzelnen Attributen im Datenmodell bestimmen. Dazu verwendet es sehr intensiv das MetaModelDAO Entity-EJB. Hier kommt es, genau wie im Query-Prozessor, unter hoher Last zu einem Fehlverhalten (Deadlocks). Dies führt dazu, dass das StatefulRSPCtrlEJB unter hoher Last nicht mehr zuverlässig die Abhängigkeiten zwischen den Attributen bestimmen kann. Das hat wiederum zur Folge, das die Übersichtsseiten nicht mehr korrekt erzeugt werden können. Trotzdem zeigt das StatefulRSPCtrlEJB ein besseres Skalierungsverhalten als das StatelessRSPCtrlEJB. Ein Grund hierfür stellt sicherlich das Zwischenspeichern von Zustandsinformationen direkt im EJB dar, denn das StatefullRSPCtrlEJB benötigt wesentlich weniger DB-Zugriffe zur Beantwortung einer Anfrage.

Abbildung 9.3: Die durchgeführten Web-Interaktionen pro Sekunde.

CachingRSPCtrlEJB Das CachingRSPCtrlEJB ist bis auf seinen Cache identisch zum StatefulRSPCtrlEJB, deswegen ergeben sich hier die gleichen Vor- und Nachteile wie beim zuvor genannten EJB. Es skaliert allerdings unter hoher Last besser als die Stateful-Variante. Offensichtlich ist dies ein positive Auswirkung des Cachens der Daten im EJB-Tier. Dies lässt vermuten, dass die Probleme (Deadlocks) im Zusammenhang mit der Datenbank bzw. dem Orion-Server nicht mit der Dauer der DB-Verbindungen oder der Menge der gelesenen Daten zusammenhängt, sondern vielmehr damit, wie häufig durch ein Enterprise-Java-Bean DB-Verbindungen auf- und wieder abgebaut werden.

Allerdings muss man beim CachingRSPCtrlEJB den zusätzlichen Speicherbedarf durch das Zwischenspeichern der Daten im EJB-Tier beachten. Die XML-Fragmente in der Datenbank können bis zu 10 KByte groß sein, dies bedeutet das bei zehn Einträgen pro Übersichtsseite etwa 300 KByte pro Client zwischengespeichert werden. Bei einer Anzahl vom 400 Clients sind das immerhin 120 MByte. Bei der technischen Ausstattung der Systeme der Messumgebung stellte dies allerdings kein Problem dar. Eine Messreihe mit einer Speicherbeschränkung auf 64 MByte für den EJB-Tiers (bzw. Orion-Server) erbrachte entsprechende Ergebnisse:Ab einer Last von 150 Simulierten-Benutzern traten Speicherfehler (OutOfMemoryException) auf.

FastLaneReader Der FastLaneReader zeigt das beste Skalierungsverhalten in diesem Test. Da er zum Lesen der XML-Fragmente direkt über eine JDBC-Verbindung auf die Datenbank zugreift und somit den EJB-Container des Orion-Servers umgeht, kommt es im Orion-Server zu weniger Problemen in Verbindung mit den Entity-Beans.

9.3.2 Leistungsfähigkeit bei Begrenzung der DB-Verbindungen

Um ein praxisbezogenes Bild des Skalierungsverhaltens unterschiedlichen Implementierungen zu erhalten, wurde eine weitere Folge von Messreihen durchgeführt. Hierzu wurde die maximale Anzahl der gleichzeitig möglichen Verbindungen eingeschränkt. Zur Umsetzung dieser Einschränkung wurde die maximale Größe des Verbindungs-Pools des Orion-Server eingeschränkt. Diese Einschränkung war leider für den FastLaneReader nicht umsetzbar, da dieser (aus technischen Gründen) seine DB-Verbindungen nicht über den Pool der Orion-Servers abwickelt. Auch auf Seite des DBVS konnte die maximale Anzahl der Verbindungen leider nicht festgelegt werden. aus diesem Grund wird der FastLaneReader in den folgenden Untersuchungen nicht weiter beachtet.

Ziel dieser Messreihen war es, zu evaluieren, ob eine der drei Implementierungen bei Einschränkung der maximalen Verbindungszahl Vorteile gegenüber den Anderen erzielen kann.

Untersuchung des Connection-Pool von Orion

Abbildung 9.4: Zeigt die Anzahl der DB-Verbindungen im Pool des Orion-Servers während eines Messlaufs mit 300 simulierten Benutzern unter Verwendung des CachingRSPCtrlEJB.

Zunächst wurden vier Messreihen durchgeführt, um die jeweilige Größe des Verbindungs-Pools des Orion-Servers zu bestimmen. Hierzu wurde das Verbindungs-Logging des Orion-Server für die Zeit dieser Messreihen aktiviert. Anhand der so erhaltenen Log-Informationen kann die Größe des Verbindungs-Pool während eines Messlaufs bestimmt werden. Da die Einträge des Verbindungs-Logs keinen Zeitstempel tragen, besteht keine Möglichkeit einen direkten zeitlichen Bezug zwischen diesen Log-Informationen und den Log-Informationen der verschiedenen Implementierungen herzustellen. Es besteht also lediglich die Möglichkeit den Verlauf der Pool-Größe während eines Messlaufs zu protokollieren.

Die Abbildungen 9.4, 9.5 und 9.6 zeigen den Verlauf der Pool-Größe während eines Messlaufs unter Verwendung der jeweils angegebenen Implementierung.

Abbildung 9.5: Zeigt die Anzahl der DB-Verbindungen im Pool des Orion-Servers während eines Messlaufs mit 300 simulierten Benutzern unter Verwendung des StatefulRSPCtrlEJB

Das StatelessRSPCtrlEJB öffnet eine DB-Verbindung immer nur für eine kurze Zeitspanne, am Ende jedes Methodenlaufs wird in jedem Fall die DB-Verbindung geschlossen. Die beiden Stateful-Implementierungen des Resultset-Prozessors hingegen halten während ihrer gesamten Lebenszeit ein eigene DB-Verbindung offen. Aus Sicht des EJB-Tiers muss

davon ausgegangen werden, dass von den Stateful-Implementierungen für n Clients n Verbindungen im Pool benötigt werden. Deswegen war zu erwarten, dass das StatelessRSPCtrlEJB deutlich weniger Verbindungen benötigt als das StatefulRSPCtrlEJB und das CachingRSPCtrlEJB.

Abbildung 9.6: Zeigt die Anzahl der DB-Verbindungen im Pool des Orion-Servers während eines Messlaufs mit 300 simulierten Benutzern unter Verwendung des StatelessRSPCtrlEJB

Wie in den Abbildungen zu sehen ist, war dies allerdings nicht der Fall. Das StatelessRSPCtrlEJB verwendet im Maximum sogar mehr DB-Verbindungen als die beiden Stateful-Implementierungen. Erklärt werden kann diese Verhalten nur dadurch, dass der Orion-Server die (Trans)Aktionen mehrerer Java-Verbindungs-Objekte über eine gemeinsame DB-Verbindung ausführt. Dies ist allerdings nur solange möglich, wie es sich um rein lesende (Trans)Aktionen handelt. Bei schreibenden Transaktionen ist ein solches Teilen der DB-Verbindungen nicht möglich. Aus diesem Grund ist auch der Bedarf an DB-Verbindungen beim StatelessRSPCtrlEJB größer, denn nur diese Implementierung des Resultset-Prozessors führt schreibende Transaktionen durch (sichern des Zustandes und sortieren der Ergebnismenge).

Im Gegensatz dazu führt das ständige Offenhalten einer DB-Verbindung (im EJB-Tier) in den beiden Stateful-Implementierungen nicht zu dem erwarteten hohen Bedarf an echten DB-Verbindungen im Verbindungs-Pool des Orion.

Es ist somit zu erwarten, dass die verschiedenen Implementierungen bei Einschränkung der maximalen Verbindungszahl ein ähnliche Verhalten an den Tag legen werden.

Weiterhin kann man den Abbildungen entnehmen, dass, bis auf wenige Ausnahmen, die Zahl der Verbindungen deutlich unter 150 liegt. Zusätzliche Messreihen ergaben, dass die Anzahl der im Pool befindlichen DB-Verbindungen im Schnitt bei allen Implementierungs-Varianten im Bereich von etwa der Hälfte der Anzahl der Simulierten-Benutzer lag.

Es lässt sich vermuten, dass eine starke Beeinträchtigung der Skalierbarkeit erst auftreten wird, wenn die maximale Anzahl der DB-Verbindungen im Pool des Orion-Servers unter die Hälfte der Anzahl der simulierten Clients fällt. Dieser Sachverhalt sollte mit den im Folgenden beschriebenen Messläufen genauer untersucht werden.

Messreihen mit eingeschränkter Poolgröße

Abbildung 9.7: Die Anzahl der durchgeführten Web-Interaktionen im Test. Durch die Kreise werden die verschiedenen Poolgrößen gekennzeichnet.

Die Abbildung 9.7 stellt die Ergebnisse von vier Testdurchläufen dar. In jedem Testdurchlauf wurden drei Messreihen durchgeführt, deren Konfiguration in jedem dieser

Testdurchläufe identisch war. Der Unterschied in den vier Testdurchläufen lag in der Konfiguration des Orion-Servers. Wie in der Grafik 9.7 angedeutet, wurde die maximale Größe des Verbindungs-Pools des Orion-Servers in den einzelnen Testdurchläufen verändert. Es wurde je ein Testdurchlauf mit einer Pool-Größe von 25, 50, 100 und 150 durchgeführt.

Die Abbildung 9.7 stellt die Summe der in den Messdurchläufen jeweils durchgeführten Web-Interaktionen dar. Mit den Kreisen werden die Messkurven jeweils eines Testdurchlaufs gekennzeichnet.

Poolgröße EJB-Typ	Minimum	Maximum	Durchschnitt
25 Verbindungen			
StatelessRSPCtrlEJB	0.58437	0.85348	0.74717
StatefulRSPCtrlEJB	0.37106	0.70044	0.53820
CachingRSPCtrlEJB	0.33542	0.65142	0.52777
50 Verbindungen			
StatelessRSPCtrlEJB	0.27517	0.69683	0.47253
StatefulRSPCtrlEJB	0.25103	0.47748	0.36392
CachingRSPCtrlEJB	0.21301	0.42971	0.32067
100 Verbindungen			
StatelessRSPCtrlEJB	0.00418	0.33769	0.17449
StatefulRSPCtrlEJB	0.04016	0.29207	0.21714
CachingRSPCtrlEJB	0.01120	0.26913	0.18426
150 Verbindungen			
StatelessRSPCtrlEJB	0.00179	0.18808	0.05415
StatefulRSPCtrlEJB	0.03651	0.23207	0.15433
CachingRSPCtrlEJB	0.00998	0.24143	0.11767

Tabelle 9.1: Fehlerraten pro Web-Interaktion der einzelnen Messreihen.

Wie bereits im vorangegangenen Abschnitt vermutet, zeigt sich zwischen den einzelnen Implementierungs-Varianten kein signifikanter Unterschied bezüglich der Zahl der durchgeführten Web-Interaktionen. Erst bei einer Poolgröße von 150 zeigen sich langsam Unterschiede zwischen den einzelnen Implementierungen. Es ist wegen der Größe des Pools allerdings wahrscheinlicher, dass diese Unterschiede (besonders bei den Messläufen mit weniger als 200 Clients) durch die in 9.3.1 beschriebenen Gründe verursacht werden, als durch die Poolgröße des Orion-Servers.

Betrachtet man die Messkurven, so könnte man zunächst von einem guten Skalierungsverhalten der verschiedenen Implementierungen ausgehen. Mit wachsender Zahl der Clients, steigt auch die Anzahl der in den jeweiligen Messläufen durchgeführten Web-Interaktionen.

Ein Vergleich der absoluten Werte der im Laufe der einzelnen Messreihen durchgeführten Web-Interaktionen mit denen aus Abschnitt 9.3.1 (siehe Abb. 9.1) lässt jedoch erkennen, dass dort ein Vielfaches an Web-Interaktionen durchgeführt wurde.

Zieht man zusätzlich noch die Tabelle 9.1 hinzu, muss man feststellen, dass die Fehlerraten in einigen Messreihen sehr hoch sind. Diese hohen Fehlerzahlen sind darin begründet, dass der Resultset-Prozessor (bzw. die Query-Engine) unter hoher Last keine neue DB-Verbindungen mehr öffnen kann und somit die Verarbeitung mit einer Fehlermeldung abbrechen muss.

Ein Pool von 25 DB-Verbindungen erweist sich selbst bei geringer Last (100 Clients) als deutlich zu gering um ein sinnvolles Arbeiten mit dem Meta-Akad System zu ermöglichen. Die durchschnittlichen Fehlerraten der einzelnen Implementierungen liegt dort im Bereich von 50% bis 70%. Sogar die minimalen Fehlerraten liegen bei der am günstigsten abscheidenden Variante, dem CachingRSPCltrEJB, noch über 30%.

Die Tabelle zeigt, dass mit steigender Poolgröße die Fehlerrate stark abnimmt. Erst bei einer Poolgröße von 150 wird eine Fehlerrate von 5-15% erreicht.

9.3.3 Mit vielen Umsortierungen

In diesem letzten Testdurchlauf sollten die Auswirkungen der unterschiedlichen Sortierstrategien der Stateless- und Stateful-Varianten des Resultset-Prozessors genauer untersucht werden.

Abbildung 9.8: Anzahl der durchgeführten Web-Interaktionen einer Web-Interaktion.

Um eine stärkere Gewichtung der Sortierroutinen zu erreichen, wurde der Quell-Code des Lastgenerators abgeändert. In dieser speziell angepassten Version, liegt die Wahrscheinlichkeit dass ein Simulierter-Benutzer die changeSortOrder-Web-Interaktion ausführt bei 50%. Zusätzlich wurden die Verteilung der Benutzertypen so geändert, dass jetzt 60% der Simulierten-Benutzer drei oder mehr Übersichtsseiten anfordern. So sollte bei gleichbleibender Last die Anzahl der von der Query-Engine durchgeführten Suchanfragen reduziert werden. Da das Ausführen einer Suchanfrage im Verhältnis zu den übrigen Interaktionen relativ viel Zeit in Anspruch nimmt, konnte durch die Reduzierung der Suchanfragen das Gewicht der übrigen Interaktionen (und somit auch das der Sortierroutinen) in den Messergebnissen gesteigert gesteigert werden.

Der FastLaneReader konnte auch hier nicht in die Messungen mit einbezogen werden, da er aus technischen Gründen nur eine fest einprogrammierte Sortiermethode beherrscht.

Abbildung 9.9: sort:sm:IpS

Die Abbildung 9.8 stellt die Anzahl der im Laufe der Messreihen durchgeführten Web-Interaktionen dar. Wie man leicht sehen kann, zeigt nun das StatelessRSPCtrlEJB ein etwas besseres Skalierungsverhalten wie das Stateful- und das CachingRSPCtrlEJB. Hier macht sich offensichtlich die Sortierroutine der zustandslosen Implementierung positiv bemerkbar. Das StatelessRSPCtrlEJB skaliert in diesem Test sogar besser als im Eingangs dieses Kapitels beschriebenen (siehe Abbildung 9.1). Ein Grund hierfür sind sicherlich die Vorteile die bei häufigen Umsortierungen aus der Speicherung und Wiederverwendung der Sortierfolgen gezogen werden können. Einen weiteren Grund für das bessere Skalierungsverhalten könnte die, bereits beschriebene, Reduzierung der Suchanfragen (der Query-Engine) darstellen. Durch diese Reduzierung sinkt automatisch die Zahl der Fehler die bei hoher Last durch die Query-Engine verursacht werden. Somit sinkt auch die Gesamtfehlerrate der Messreihe. Der Vergleich der Abbildungen 9.9 und 9.2 zeigt, dass die Fehlerrate des StatelessRSPCtrlEJB in beiden Fällen zunächst relativ gering bleibt (unter 5%). Ab einer Last von 250 Clients steigt die Fehlerrate in beiden Abbildungen stark an. In der Messreihe aus Abschnitt 9.3.1 steigt sie allerdings wesentlich schneller als hier. Am Ende der

Messreihe erreicht die Fehlerrate des Tests aus Abschnitt 9.3.1 einen Wert von fast 40% und die der Messreihe dieses Abschnitts nur etwa 25%.

Weiterhin fällt auf, dass die beiden Stateful-Implementierungen wesentlich schlechter skalieren als im Eingangs beschriebenen Test. Hier zeigt sich deutlich der Nachteil dieser Sortierroutinen, denn diese müssen aufwendige Join Operationen zum Sortieren der Ergebnistabelle bei jeder Umsortierung wiederholen.

Kapitel 10

Fazit und Ausblick

10.1 Fazit

Ziel dieser Diplomarbeit war es verschiedene Skalierungskonzepte in datenintensiven, mehrschichtigen und verteilten Anwendungsarchitekturen vorzustellen und quantitativ zu evaluieren.

Zu Beginn wurden die Grundlagen zum Verständnis von verteilten und mehrschichtigen Software-Architekturen erarbeitet. Zunächst wurden die Begriffe „datenintensiv“ und „verteilte Software-Systeme“ präzisiert. Daran anschließend wurden verschiedene Techniken zur Skalierung von Software-Systemen vorgestellt. Im zweiten Teil dieses Kapitels wurden verschiedene Architektur-Konzepte vorgestellt und unter dem Gesichtspunkt ihrer Skalierbarkeit bewertet. Besonders mehrschichtige Architektur-Konzepte wurden an dieser Stelle hervorgehoben.

Spricht man von mehrschichtigen Software-Systemen, so sind in der Regel komponentenbasierte Systeme gemeint. Deswegen wurde im nächsten Kapitel der Komponentenbegriff genauer erläutert und eine ausführliche Definition hierfür gegeben. Da im Besonderen Skalierungskonzepte in Bezug auf zustandslose bzw. zustandsbehaftete Komponenten untersucht werden sollten, wurde in diesem Kapitel zusätzlich der Zustandsbegriff eingeführt. Dabei wurde eine Definition für Zustände von Komponenten gegeben und erläutert was unter zustandslosen bzw. zustandsbehafteten Komponenten zu verstehen ist.

Da die in den ersten Kapiteln vorgestellten Konzepte in einer praxisbezogen Umsetzung evaluiert werden sollten, wurde nun die Java 2 Enterprise Edition als Entwicklungsplattform vorgestellt. Es wurde dabei das J2EE Architektur-Modell vorgestellt und in die Architektur-Konzepte der bereits Vorgestellten eingeordnet. Ebenso wurde das J2EE zugrundeliegende Komponenten-Modell vorgestellt und dabei die verschiedenen Komponenten-Typen erläutert. Die Stateless- bzw. Stateful-Session-EJBs wurden hierbei als J2EE Äquivalent der bereits vorgestellten zustandslosen und zustandsbehafteten

Komponenten identifiziert.

Die Evaluierung der verschiedenen Skalierungskonzepte sollte im Rahmen des Meta-Akad Forschungsprojekts vorgenommen werden. Aus diesem Grund wurde das Meta-Akad Projekt hier ausführlich vorgestellt. Nach einer kurzen Übersicht über das Gesamtprojekt, wurden die zur Realisierung des Systems eingesetzten Entwurfskonzepte vorgestellt. Hierbei wurde insbesonders das Komponenten-Modell des Meta-Akad Systems sowie die Funktion der einzelnen Komponenten betrachtet. Teil dieses Komponenten-Modells war es die Resultset-Prozessor Komponente zum Einen als zustandsbehaftete- und zum Anderen als zustandslose Komponente zu realisieren. In einem gesonderten Abschnitt wurden die erwarteten Probleme im Zusammenhang mit dieser zustandslosen Implementierung erläutert und verschiedene Konzepte vorgestellt um diese Probleme zu lösen.

Direkt im Anschluss daran, wurden verschiedene, zur Evaluierung des Skalierungsverhaltens wichtige, Implementierungsdetails der Query-Engine und der verschiedenen Resultset-Prozessor Varianten erläutert. Um die Abläufe im Meta-Akad System zu verdeutlichen, wurde als Beispiel die Abarbeitung einer Suchanfrage ausführlich beschrieben.

Um die Evaluierung der verschiedenen Entwurfs-Konzepte (in Form der verschiedenen Resultset-Prozessor Versionen) vornehmen zu können, musste nun ein Messverfahren erarbeitet werden. Hierzu wurden in Anlehnung an den TPC-W Benchmark eine Reihe von Konzepten entworfen und zu messende Größen identifiziert. Im zweiten Teil diese Kapitels wird erläutert, wie diese Konzepte in Form eines Lastgenerator umgesetzt wurden. Dabei werden auch einige technische Probleme bei der Entwicklung des Lastgenerator erläutert und entsprechende Lösungen vorgestellt. Anschließend werden die durch den Lastgenerator erzeugten Messergebnisse konzeptionell vorgestellt und beschrieben wie mittels eines eigens hierfür erstellten Shell-Skriptes aus diesen Rohdaten auswertbare, aggregierte Messergebnisse erzeugt werden können.

Nun konnte die Evaluierung des Meta-Akad Systems und der verschiedenen Implementierungen des Resultset-Prozessors vorgenommen werden. Dabei stellte sich heraus, dass die zustandslose Variante des Resultset-Prozessors deutlich schlechter skaliert als die zustandsbehafteten Versionen. Dies lag besonders an der hohen Belastung des DBVS durch die vielen Operationen zum Sichern und Wiederherstellen von Zustandsinformationen durch den (zustandslosen) Resultset-Prozessor. Dies führte sogar dazu, dass andere Komponenten (etwa die Query-Engine) die häufig auf das DBVS zugreifen hierbei ebenfalls ein schlechteres Skalierungsverhalten zeigten als im Zusammenspiel mit den zustandsbehafteten Varianten des Resultset-Prozessors. Im Gegenzug dazu blieben die beim Einsatz der zustandsbehafteten Versionen des Resultset-Prozessors erwarteten negativen Aspekte, wie etwa eine sehr hohe Zahl an gleichzeitig benötigten DB-Verbindungen, aus. Die Gründe hierfür liegen im Verbindungs-Management des Orion-Servers. Dieser verarbeitet die Transaktionen mehrerer (lesender) JDBC-Verbindungen über eine gemeinsame reale DB-Verbindung. Ein allgemeines Problem im Zusammenhang mit dem

Orion-Server stellten auch die durch diesen automatisch generierten Entity-EJBs dar. Hier kam es häufig bei nur lesenden Transaktionen zu Deadlock Problemen (möglicherweise exklusive Lesesperren). Leider ist das interne Verhalten des Orion-Servers nicht (öffentlich zugänglich) dokumentiert, so dass diese Problematiken nicht weiter untersucht werden konnten.

Zusammenfassend lässt sich sagen, dass die in dieser Diplomarbeit erzielten Ergebnisse nur bedingt auf allgemeine Skalierungskonzepte von zustandslosen bzw. zustandsbehafteten Komponenten übertragen werden können. Es konnte zwar festgestellt werden, dass der hohe Kommunikationsaufwand der zustandslosen Komponenten in der Tat sehr schnell die Vorteile dieses Komponenten-Konzeptes zunichte machen kann. Allerdings lassen sich genaue Aussagen auf Grund des nicht dokumentierten Verhaltens des Orion-Servers leider nicht machen.

10.2 Ausblick

Um die beschriebenen Probleme bei der Skalierbarkeit der vorgenommenen Implementierung zu lösen, wären verschiedene Ansätze denkbar.

Zur Reduzierung der Zugriffsprobleme durch die automatisch erstellten Klassen der Entity-EJBs, könnte man dazu übergehen diese von Hand zu implementieren. Dadurch würden man volle Kontrolle über das Transaktionsverhalten dieser EJBs erlangen. So sollten sich die auftreten Probleme (Deadlocks) bei lesenden Transaktionen vollständig vermeiden lassen.

Ein weiterer Weg die Problematiken mit den automatisch generierten Klassen zu umgehen, wäre die Migration des Meta-Akad Systems auf einen anderen Application-Server. Hierzu wäre allerdings eine genaue Inspektion aller Teile des Meta-Akad Systems notwendig, um die nicht standardkonformen Teillösungen zu identifizieren (und gegebenenfalls anzupassen).

Um die Zahl der DB-Zugriffe durch die zustandslose Variante des Resultset-Prozessors zu reduzieren, sind ebenfalls verschiedene Lösungen denkbar. So könnte man um das „Backend“ Datenbanksystem zu entlasten, zur Sicherung der Zustandsinformationen (Offset, Suchmethode, etc.) ein zweites DBVS einsetzen. Dadurch sollte die Belastung des Backend DBVS beim Einsatz des zustandslosen Resultset-Prozessors nicht mehr höher sein als bei den zustandsbehafteten Varianten.

Eine weitere Möglichkeit diese Zugriffsprobleme zu reduzieren, wäre das Speichern der Zustandsinformationen in den Client-Komponenten. Würde man die Kommunikation zwischen den Komponenten eher „nachrichten-orientiert“ durchführen, etwa durch den Austausch von XML-Dokumenten (oder Java-Objekten), so könnten die benötigten Zustandsinformationen direkt, beim Aufruf einer Methode durch den Client, mit übergeben werden. Umgekehrt könnten alle Änderungen der Zustandsinformationen als Teil des Rückgabewertes wieder an den Client übermittelt werden. Dadurch ließe sich das Sichern und

Wiederherstellen des Zustandes mit Hilfe eines DBVS völlig vermeiden.

Die Leistungsfähigkeit des Meta-Akad System ließe sich auch dadurch erhöhen, dass versucht man die bereits erzeugten Ergebnistabellen bei Folgeanfragen wieder zu verwenden. Relativ einfach umzusetzen, wäre vermutlich der Fall, das eine Suchanfrage durch einen Client verfeinert wird. Startet etwa ein Client nach einer erfolgreichen Suchanfrage eine weitere, indem er den vorangegangenen Suchstring erweitert (also die Menge der Suchergebnisse einschränkt), so würde es genügen eine Suche auf der zuvor erzeugten Ergebnistabelle durchzuführen. Dieses Konzept ließe sich auch noch erweitern, indem man zur Optimierung einer Suchanfrage nicht nur die durch einen Client selbst erzeugten Ergebnistabellen, sondern alle Ergebnistabellen heranzieht. Zur Umsetzung einer solchen Strategie würden sich die in 6.3.1 beschriebenen MQTs bzw. MViews bestens eignen.

Eine weitere Möglichkeit das Skalierungsverhalten der hier vorgestellten Komponenten zu verbessern wäre ein zusätzlicher Cache im Middle-Tier. Beispielsweise beschreibt C. Mohan in [Moh02], wie durch einfache Erweiterungen des DB2 UDB Datenbankverwaltungssystems dieses in einen, wie er es nennt, *DBCache* umgewandelt werden kann. Dieser DBCache wird vom Application-Server im Middle-Tier bereitgestellt (deployed) und kann von den übrigen Applikationen genau wie auf ein normales Backend-DBVS angesprochen werden. DBCache kümmert sich dann um alle mit dem Cachen von Daten im Middle-Tier einhergehenden Problematiken (z.B.: Konsistenz, Aktualität, etc.).

Anhang A

Lastgenerators

A.1 Konfiguration des Lastgenerators

Auszug aus einem XML-Konfigurationsfile für den Lastgenerator. Ausgehend von einem solchen Konfigurationsfile, kann mit dem Skript genConfig eine Reihe von Konfigurationsdateien erzeugt werden. Diese sind bis auf die Zahl der zu simulierenden Clients identisch.

```
\input{/home/gauss/CONFIGS/Config_Defaults/config_base.xml}
<?xml version="1.0" standalone="yes"?>

<!--<!DOCTYPE RSP:Benchmark SYSTEM "./Benchmark_Config.dtd"> -->

<RSP:Benchmark xmlns:RSP="rspclient-config/" beanType="stateless"
name="/tmp/MetaAkad-Tests/XXX_stateless">

<RSP:User numberOfUsers="XXX" runTimeMin="20"/>
<RSP:UserType number="1" probability="48"/>
<RSP:UserType number="2" probability="23"/>
<RSP:UserType number="3" probability="10"/>
<RSP:UserType number="4" probability="9"/>
<RSP:UserType number="5" probability="10"/>
<RSP:PageSize size="10"/>
<!-- Zeitangaben jeweils in ms. -->
<RSP:ThinkTime upperBoundary="45000" lowerBoundary="15000"/>
<RSP:StartDelay upperBoundary="300000" lowerBoundary="1000"/>

<!-- Ab hier folgen nur noch Suchbegriffe. Die Anzahl der zu erwartenden
Treffer je Suchwort steht als Kommentar am Ende jeder Zeile.-->

      <RSP:SearchString>didaktische</RSP:SearchString><!-- 476 -->
      <RSP:SearchString>math</RSP:SearchString><!-- 1366 -->
      <RSP:SearchString>quanten</RSP:SearchString><!-- 554 -->
```

```
      <RSP:SearchString>themen</RSP:SearchString><!-- 1271 -->
      <RSP:SearchString>nach</RSP:SearchString><!-- 595 -->
      <RSP:SearchString>function</RSP:SearchString><!-- 70 -->
      <RSP:SearchString>page</RSP:SearchString><!-- 155 -->
      <RSP:SearchString>design</RSP:SearchString><!-- 53 -->
      <RSP:SearchString>vorlesung</RSP:SearchString><!-- 444 -->
      <RSP:SearchString>material</RSP:SearchString><!-- 538 -->
      <RSP:SearchString>didaktik</RSP:SearchString><!-- 487 -->
      <RSP:SearchString>heme</RSP:SearchString><!-- 1277 -->
      <RSP:SearchString>relativitätstheorie</RSP:SearchString><!-- 161 -->
      <RSP:SearchString>teilchen</RSP:SearchString><!-- 163 -->
      <RSP:SearchString>quantenstatistik</RSP:SearchString><!-- 168 -->
      <RSP:SearchString>time</RSP:SearchString><!-- 75 -->
      <RSP:SearchString>modern</RSP:SearchString><!-- 45 -->
      <RSP:SearchString>phys</RSP:SearchString><!-- 1459 -->
      <RSP:SearchString>eine</RSP:SearchString><!-- 1557 -->
      <RSP:SearchString>sicht</RSP:SearchString><!-- 45 -->
      <RSP:SearchString>geschichte</RSP:SearchString><!-- 613 -->
      <RSP:SearchString>theorie</RSP:SearchString><!-- 1191 -->
      <RSP:SearchString>lehre</RSP:SearchString><!-- 473 -->
      <RSP:SearchString>physik</RSP:SearchString><!-- 1431 -->
      <RSP:SearchString>einem</RSP:SearchString><!-- 186 -->
      <RSP:SearchString>graph</RSP:SearchString><!-- 657 -->
      <RSP:SearchString>allgemeine</RSP:SearchString><!-- 1161 -->
      <RSP:SearchString>band</RSP:SearchString><!-- 197 -->
      <RSP:SearchString>rainer</RSP:SearchString><!-- 33 -->
      <RSP:SearchString>allgemeines</RSP:SearchString><!-- 240 -->
      <RSP:SearchString>partielle</RSP:SearchString><!-- 94 -->
      <RSP:SearchString>beitrag</RSP:SearchString><!-- 246 -->
      <RSP:SearchString>infinitesimalrechnung</RSP:SearchString><!-- 12 -->
      <RSP:SearchString>geometrie</RSP:SearchString><!-- 250 -->
      <RSP:SearchString>them</RSP:SearchString><!-- 1421 -->
      <RSP:SearchString>method</RSP:SearchString><!-- 487 -->
      <RSP:SearchString>mathematik</RSP:SearchString><!-- 1305 -->
      <RSP:SearchString>topics</RSP:SearchString><!-- 52 -->
      <RSP:SearchString>lehrmaterial</RSP:SearchString><!-- 479 -->
      <RSP:SearchString>near</RSP:SearchString><!-- 277 -->
      <RSP:SearchString>kontinuumsmechanik</RSP:SearchString><!-- 282 -->
      <RSP:SearchString>mathe</RSP:SearchString><!-- 1361 -->
      <RSP:SearchString>mechanik</RSP:SearchString><!-- 491 -->
      <RSP:SearchString>site</RSP:SearchString><!-- 53 -->

</RSP:Benchmark>
```

A.2 genConfig

Generiert die Konfigurationsdateien für eine Messreihe ausgehend von einer Basiskonfiguration (siehe A.1). Das Skript nimmt drei Parameter auf der Kommandozeile entgegen. Der Erste legt die minimale Anzahl der zu simulierenden Clients fest. Der Zweite bestimmt die „Schrittweite“ der Messreihe (um welche Größe jeweils die Clientszahl erhöht werden soll). Der dritte Parameter legt die maximale Clientanzahl fest.

```
#!/bin/bash

((START=$1))
((TIME=$2))
((END=$3))
((FOLGE=1))

for i in *.xml; do
echo processing $i
((j=$START))
while ((j <= END)); do
SEDCOMM=s\\XXXX\\$j\\
cat $i | sed -e $SEDCOMM > $FOLGE-$j-$i
((j = $j + $TIME))
((FOLGE= $FOLGE +1))
done
done
```

A.3 log2plot

Erzeugt eine Datei mit den aggregierten Messergebnissen aus den Log-Dateien der einzelnen Messläufe einer Messreihe. Dieses Skript nimmt als Parameter den Namer der Datei, dessen Inhalt ausgewertet werden soll. Das Ergebnis der Auswertung wird auf der Konsole ausgegeben.

```
#!/bin/bash

INPUT=$1

#leerzeilen und zeilen mit ungültigem format löschen
#cat $INPUT | grep -v "^[ ]*$" > $INPUT.temp1
#cat $INPUT | sed '/^$/d' | sed '/^[^0-9].*:[^0-9].*$/d'  | sed '/^$/d' > $INPUT.temp1

echo \#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#\#
echo \#Working on file $INPUT

# in min
```

```
((STARTOFFSET=5))
((ENDOFFSET=25))

#umrechnen im ms
((STARTOFFSET = STARTOFFSET * 60 * 1000))
((ENDOFFSET = ENDOFFSET * 60 * 1000))

#fehler aus $INPUT eliminieren
gawk -F: '{
if (( $5 != 0 )&&($6 != 0)) {
print $0;
}
}' $INPUT > $INPUT.temp2
#cat $INPUT.temp2 | grep -v "^[ ]*$" > $INPUT.temp3
cat $INPUT.temp2 | sed '/^$/d' | sed 's/^[^0-9].*:[^0-9].*$//' \
|sed '/^$/d'|sed '/.*:0:0:0.*/d'| sed '/.*Exception.*/d' > $INPUT.temp3

#Zeitintervall bestimmen aus dem die Werte entnommen werden
MIN=`gawk -F : 'BEGIN{ minval = 0 } ; {
if (( $5 < minval )||(minval == 0)) {
minval = $5;
}
}; END {print minval}' $INPUT.temp3`
echo \#Start of first interaction $MIN

((START = $MIN + $STARTOFFSET))
((END = $MIN +$ENDOFFSET))
echo \#Results between $START and $END will be used for calculations.

#Alle Interaktionen mit Fehler:
ECOUNT=`cat $INPUT | grep -i -c "0:0:0"`
#Alle Interaktionen ohne Fehler
NECOUNT=`cat $INPUT | grep -v -i -c "0:0:0"`
#Alle Interaktionen mit deadlockfehler
DCOUNT=`cat $INPUT | grep -i -c "deadlock"`
#Alle Interaktionen insgesammt
((TCOUNT=$ECOUNT+$NECOUNT))

#Verhältnis von Fehler zu Interaktion
ETCOUNT=`echo "scale=5; $ECOUNT/$TCOUNT" | bc`
DTCOUNT=`echo "scale=5; $DCOUNT/$TCOUNT" | bc`

#Ausgabestring der Fehlerwerte
ERR=$TCOUNT" "$ECOUNT" "$ETCOUNT" "$DCOUNT" "$DTCOUNT
```

```
echo \#
echo \#1:  NoC = Number of Clients
echo \#2:  TfI = Total time for Interactios
echo \#3:  NoI = Number of Interactios
echo \#4:  ATI = Avg time for a single Interaction
echo \#5:  IpS = Interactions per second
echo \#6:  TC = Total count of Interactions
echo \#7:  E = Absolute errorcount
echo \#8:  EpI = Errors per Interactions
echo \#9:  D = Total count of Deadlocks in DB
echo \#10:  DpI = Deadlocks per Interaction

#Bestimmt die Zahl der Interaktionen,Interaktionen pro Sekunde und die
#durchschnittliche Antwortzeit einer Interaktion (jeweils im Messintervall)
AVG=‘gawk -F:  ’{
if (( min <= $5 )&&( max >= $5)) {
sum += $7;
count += 1;
}
if ( maxval < $1 ) {
maxval = $1;
}
}; END {
if ( count == 0 ) {
printf("FEHLER: keine Daten. %d %d %d ",maxval+1,sum,count);
} else {
printf("%d %d %d %f %f",maxval+1,sum,count,
sum/count,count/((max-min)/1000));
}
}’ min=$START max=$END $INPUT.temp3‘
echo \#NoC TfI NoI ATI IpS TC E EpI D DpI
echo $AVG $ERR

rm -f $INPUT.temp*
```

A.4 gnuplot

Ein beispielhaftes Skripts für das Programm gnuplot. Hiermit können aus den aggregierten Messdaten die Messkurven erzeugt werden. Dieses Skript kann in gnuplot geladen werden und erzeugt dann eine Bildschirmausgabe der Messwerte.

```
#!/usr/bin/gnuplot -persist
#
#
#     G N U P L O T
```

```
#     Version 3.7 patchlevel 2
#     last modified Sat Jan 19 15:23:37 GMT 2002
#     System: Linux 2.4.19
#
#     Copyright(C) 1986 - 1993, 1998 - 2002
#     Thomas Williams, Colin Kelley and many others
#
#     Type ‘help‘ to access the on-line reference manual
#     The gnuplot FAQ is available from
#     http://www.gnuplot.info/gnuplot-faq.html
#
#     Send comments and requests for help to <info-gnuplot@dartmouth.edu>
#     Send bugs, suggestions and mods to <bug-gnuplot@dartmouth.edu>
#
# set terminal x11
# set output
set noclip points
set clip one
set noclip two
set bar 1.000000
set border 31 lt -1 lw 1.000
set key title "RSPCtrl Varianten"
set nolabel
set noarrow
set nolinestyle
set nologscale
set offsets 0, 0, 0, 0
set pointsize 1
set encoding default
set view 60, 30, 1, 1
set samples 100, 100
set isosamples 10, 10
set surface
set clabel '%8.3g'
set mapping cartesian
set nohidden3d
set cntrparam order 4
set cntrparam linear
set cntrparam levels auto 5
set cntrparam points 5
set size ratio 0 1,1
set origin 0,0
set data style points
set function style lines
set xzeroaxis lt -2 lw 1.000
```

```
set x2zeroaxis lt -2 lw 1.000
set yzeroaxis lt -2 lw 1.000
set y2zeroaxis lt -2 lw 1.000
set tics in
set ticslevel 0.5
set ticscale 1 0.5
set nox2tics
set noy2tics
set title "" 0.000000,0.000000  ""
set timestamp "" bottom norotate 0.000000,0.000000  ""
set rrange [ * : * ] noreverse nowriteback  # (currently [-0.00000:10.0000] )
set trange [ * : * ] noreverse nowriteback  # (currently [-5.00000:5.00000] )
set urange [ * : * ] noreverse nowriteback  # (currently [-5.00000:5.00000] )
set vrange [ * : * ] noreverse nowriteback  # (currently [-5.00000:5.00000] )
set xlabel "Anzahl der simulierten Clients" 0.000000,0.000000  ""
set x2label "" 0.000000,0.000000  ""
set timefmt "%d/%m/%y\n%H:%M"
set xrange [ * : * ] noreverse nowriteback  # (currently [-10.0000:10.0000] )
set x2range [ * : * ] noreverse nowriteback  # (currently [-10.0000:10.0000] )
set ylabel "Gesamtzahl der Interaktionen" 0.000000,0.000000  ""
set y2label "" 0.000000,0.000000  ""
set yrange [ * : * ] noreverse nowriteback  # (currently [-10.0000:10.0000] )
set y2range [ * : * ] noreverse nowriteback  # (currently [-10.0000:10.0000] )
set zlabel "" 0.000000,0.000000  ""
set zrange [ * : * ] noreverse nowriteback  # (currently [-10.0000:10.0000] )
set zero 1e-08
set locale "C"
plot "caching.plot" using 1:6 title "CachingEJB" with lines,
"fast.plot" using 1:6 title "FastLaneReader" with lines,
"ful.plot" using 1:6 title "StatefulRSPCtrlEJB" with lines,
"less.plot" using 1:6 title "StatelessRSPCtrlEJB" with lines
#    EOF
```

Anhang B

RSPCtrl - Interface

Das Home- und das Remote-Interface des Resultset-Prozessors.

B.1 Remote-Interface

```
metabase.resultprocessor.ejb.rspctrl;

import java.util.*;
import javax.ejb.*;
import java.rmi.RemoteException;

import metabase.queryengine.beans.queryresult.*;
import metabase.common.beans.*;

/**
 * RSPCtrl defines the remote interface to a
 * session bean for getting access to pregenerated
 * results, which are provided by the query engine.
 * Because of its possible statelessness (there are
 * two different implementations) it is neccessary
 * to provide the result set proxy which is also
 * created by the query engine. The queries and their
 * results made by the here defined methods apply
 * to the result set only.
 *
 * @see metabase.queryengine.ejb.qectrl
 * @see metabase.queryengine.beans.queryresult.RSHandle
 * @see metabase.resultprocessor.beans.rpoutput.RPTuple
 *
 */

public interface RSPCtrl extends EJBObject
```

```
{
    /**
     * Returns the number of learningresources which have
 * been selected by the query
     *
     * @param handle represents the query and is provided
 * by the query engine
     * @return Count of qualifying learning resources
     */
    public long getHits(RSHandle handle)
      throws InvalidResultSetHandleException, RemoteException;

    /**
     * Returns the n most commonly used subjects. Throws
 * NoSubjectsException if no Subjects can be found.
     *
     * @param handle represents the query and is provided by the
 * query engine
     * @param quantity returns the n most commonly used subjects
     * @return Collection of String objects with subject description
     *
     */
    public Collection getSubjects(RSHandle handle, long quantity)
      throws InvalidResultSetHandleException, RemoteException,
NoSubjectsException;

    /**
     * Returns the n most commonly used classification entries.
 * Throws NoSubjectsException if no ClassificationEntries can be found.
     *
     * @param handle represents the query and is provided by the query engine
     * @param quantity returns the n most commonly used classification entries
     * @return Collection of ClassificationEntry objects (Entity EJBs!)
     * @see metabase.classification.ejb
     */
    public Collection getClassificationEntries(RSHandle handle, long quantity)
      throws InvalidResultSetHandleException, InvalidResultSetHandleException,
RemoteException, NoSubjectsException;

    /**
     * Returns the first n XML fragment of the resultset
     *
     *@param handle represents the query and is provided by the query engine
     *@param quantity returns the first n entries
     *@param transformationscheme is a scheme for the xalan xslt
```

```
 * processor to transfor the XML fragments
     *
     *transformationscheme can be set null to get unmodified XML fragments
     */
    public String getFirstResultSetEntriesAsXML(RSHandle handle, long quantity,
String transformationscheme) throws InvalidResultSetHandleException,
RemoteException;

    /**
     * Returns all XML fragments of the resultset
     *
     *@param handle represents the query and is provided by the query engine
     *@param transformationscheme is a scheme for the xalan xslt processor
 *to transfor the XML fragments
     *
     *transformationscheme can be set null to get unmodified XML fragments
     */
    public String getAllResultSetEntriesAsXML(RSHandle handle,
String transformationscheme)
      throws InvalidResultSetHandleException, RemoteException;

    /**
     * Returns the next n XML fragments of the resultset
     *
     *@param handle represents the query and is provided by the query engine
     *@param quantity returns the first n entries
     *@param transformationscheme is a scheme for the xalan xslt processor to
 *transfor the XML fragments
     *
     *transformationscheme can be set null to get unmodified XML fragments
     */
    public String getNextResultSetEntriesAsXML(RSHandle handle, long quantity,
String transformationscheme) throws InvalidResultSetHandleException,
RemoteException;

    /**
     * Returns the first n XML fragment of the resultset
     *
     *@param handle represents the query and is provided by the query engine
     *@param quantity returns the first n entries
     *
     *Is identical to a getFirstResultSetEntriesAsXML(handle, quantity, null)
 *call.
     */
    public String getFirstResultSetEntriesAsXML(RSHandle handle, long quantity)
```

```
      throws InvalidResultSetHandleException, RemoteException;

    /**
     * Returns all XML fragments of the resultset
     *
     *@param handle represents the query and is provided by the query engine
     *
     *Is identical to a getAllResultSetEntriesAsXML(handle, null) call.
     */
    public String getAllResultSetEntriesAsXML(RSHandle handle)
      throws InvalidResultSetHandleException, RemoteException;

    /**
     * Returns the next n XML fragments of the resultset
     *
     *@param handle represents the query and is provided by the query engine
     *@param quantity returns the first n entries
     *
     *Is identical to a getNextResultSetEntriesAsXML(handle, quantity, null)
 *call.
     */
    public String getNextResultSetEntriesAsXML(RSHandle handle, long quantity)
      throws InvalidResultSetHandleException, RemoteException;

    /**
     * Changes the current sort method and resets the resultset cursor.
     * Sort information is pregenerated and reused, i.e. queries haven't been
     * executed more than once.
     */
    public void setSortMethod(RSHandle handle, String method)
      throws InvalidResultSetHandleException,  MethodNotSupportedException,
RemoteException;

    /**
     * Gives information about the supported sort methods. Returns a
 * numeration of string objs.
     *
     * For the stateless version there are three (resp. four) different
 * supported sort method:
     *  <ul>
     *    <li>match_asc,match_desc      (Wortübereinstimmung - Auf- bzw. Absteigend)
     *    <li>general_asc,general_desc  (Allgemeines zuerst - Auf- bzw. Absteigend)
     *    <li>special_asc,special_desc  (Spezielles zuerst - Auf- bzw. Absteigend)
     *    <li>new_asc,new_desc          (Neues zuerst - Auf- bzw. Absteigend)
     *  </ul>
```

```
     *
     * @return Collection of String objects which include the sort method names
     */
    public Collection getSupportedSortMethod() throws RemoteException;

    /**
     * Gets the offset for the resultset cursor.
     */
    public long getOffset(RSHandle handle) throws InvalidResultSetHandleException,
RemoteException;

    /**
     * Sets the offset for the resultset cursor.
     */
    public void setOffset(RSHandle handle, long Offset)
throws InvalidResultSetHandleException,  RemoteException;

    /**
     * Deletes the resultset identified by the handle.
     *@handle Contians the id of the resultset to delete.
     */
    public void releaseResultSet(RSHandle handle)
throws InvalidResultSetHandleException,  RemoteException;

    /**
     * For internal use only.
     * Gives a list of possible search-strings, generated from title,
 * subject_values and name.
     * @deprecated
     */
    public ArrayList getSuchWoerter() throws RemoteException;
}
```

B.2 Home-Interface

```
ackage metabase.resultprocessor.ejb.rspctrl;

import java.rmi.RemoteException;
import javax.ejb.*;

/**
 * The home of the stateful and stateless Controller Bean
 * to build a facade while result set processing.
 * In contrast to the stateless session bean it
```

```
 * provides more and especially other sort methods.
 */
public interface RSPCtrlHome extends EJBHome
{
public RSPCtrl create() throws CreateException, RemoteException;
}
```

Anhang C

Messergebnisse

C.1 Unlimitierte DB-Verbindungen

Abbildung C.1: Keine Begrenzung des DB-Verbindungs-Pools. Absoulute Fehleranzahl.

Abbildung C.2: Keine Begrenzung des DB-Verbindungs-Pools. Durchschnittliche Antwortzeit pro Web-Interaktion.

Abbildung C.3: Keine Begrenzung des DB-Verbindungs-Pools. Deadlockfehler pro Web-Interaktion.

Abbildung C.4: Keine Begrenzung des DB-Verbindungs-Pools. Fehler pro Web-Interaktion.

Abbildung C.5: Keine Begrenzung des DB-Verbindungs-Pools. Web-Interaktionen pro Sekunde.

C.2 Maximal 25 DB Verbindungen

Abbildung C.6: DB-Verbindungs-Pool auf 25 begrenzt. Absolute Fehleranzahl.

Abbildung C.7: DB-Verbindungs-Pool auf 25 begrenzt. Durchschnittliche Antwortzeit pro Web-Interaktion.

Abbildung C.8: DB-Verbindungs-Pool auf 25 begrenzt. Deadlockfehler pro Web-Interaktion.

Abbildung C.9: DB-Verbindungs-Pool auf 25 begrenzt. Fehler pro Web-Interaktion.

Abbildung C.10: DB-Verbindungs-Pool auf 25 begrenzt. Web-Interaktionen pro Sekunde.

C.3 Maximal 50 DB Verbindungen

Abbildung C.11: DB-Verbindungs-Pool auf 50 begrenzt. Absolute Fehleranzahl.

Abbildung C.12: DB-Verbindungs-Pool auf 50 begrenzt. Durchschnittliche Antwortzeit pro Web-Interaktion.

Abbildung C.13: DB-Verbindungs-Pool auf 50 begrenzt. Deadlockfehler pro Web-Interaktion.

Abbildung C.14: DB-Verbindungs-Pool auf 50 begrenzt. Fehler pro Web-Interaktion.

Abbildung C.15: DB-Verbindungs-Pool auf 50 begrenzt. Web-Interaktionen pro Sekunde.

C.4 Maximal 100 DB Verbindungen

Abbildung C.16: DB-Verbindungs-Pool auf 100 begrenzt. Absolute Fehleranzahl.

Abbildung C.17: DB-Verbindungs-Pool auf 100 begrenzt. Durchschnittliche Antwortzeit pro Web-Interaktion.

Abbildung C.18: DB-Verbindungs-Pool auf 100 begrenzt. Deadlockfehler pro Web-Interaktion.

Abbildung C.19: DB-Verbindungs-Pool auf 100 begrenzt. Fehler pro Web-Interaktion.

Abbildung C.20: DB-Verbindungs-Pool auf 100 begrenzt. Web-Interaktionen pro Sekunde.

C.5 Maximal 150 DB Verbindungen

Abbildung C.21: DB-Verbindungs-Pool auf 150 begrenzt. Absolute Fehleranzahl.

Abbildung C.22: DB-Verbindungs-Pool auf 150 begrenzt. Durchschnittliche Antwortzeit pro Web-Interaktion.

Abbildung C.23: DB-Verbindungs-Pool auf 150 begrenzt. Deadlockfehler pro Web-Interaktion.

Abbildung C.24: DB-Verbindungs-Pool auf 150 begrenzt. Fehler pro Web-Interaktion.

Abbildung C.25: DB-Verbindungs-Pool auf 150 begrenzt. Web-Interaktionen pro Sekunde.

C.6 Maximal 200 DB Verbindungen

Abbildung C.26: DB-Verbindungs-Pool auf 200 begrenzt. Absolute Fehleranzahl.

Abbildung C.27: DB-Verbindungs-Pool auf 200 begrenzt. Durchschnittliche Antwortzeit pro Web-Interaktion.

Abbildung C.28: DB-Verbindungs-Pool auf 200 begrenzt. Deadlockfehler pro Web-Interaktion.

Abbildung C.29: DB-Verbindungs-Pool auf 200 begrenzt. Fehler pro Web-Interaktion.

Abbildung C.30: DB-Verbindungs-Pool auf 200 begrenzt. Web-Interaktionen pro Sekunde.

C.7 Sortier Test

Abbildung C.31: Sortier Test. Absolute Fehleranzahl.

Abbildung C.32: Sortier Test. Durchschnittliche Antwortzeit pro Web-Interaktion.

Abbildung C.33: Sortier Test. Deadlockfehler pro Web-Interaktion.

Abbildung C.34: Sortier Test. Fehler pro Web-Interaktion.

Abbildung C.35: Sortier Test. Web-Interaktionen pro Sekunde.

Literaturverzeichnis

[BB92] Breutmann and Burkhardt. *Objektorientierte Systeme - Grundlagen - Werkzeuge - Einsatz*. Carl Hanser Verlag, München, Wien, 1992.

[BRRJ99] Grady Booch, Jim Rumbaugh, James Rumbaugh, and Ivar Jacobson. *The UML Modeling Language User Guide*. Addison-Wesley, Harlow (u.a.), 1999.

[Buß03] Axel Bußmann. Gnuplot - kurzanleitung. http://www.we.fh-osnabrueck.de/fbwe/vorlesung/edv2/gplot/gplot.html, 2003.

[Cou99] Transaction Processing Performance Council. TPC Benchmark W (Web Commerce) Draft Specification Revision D-5.1. Technical report, TPC, San Jose, CA 96112-6311, USA, 1999.

[Cra03] Dick Crawford. gnuplot - an interactive plotting program. http://www.ucc.ie/gnuplot/gnuplot.html, 2003.

[Fla00] David Flanagan. *Java in a Nutshell*. O'Reilly, Köln, 2000.

[Fle02] Marcus Flehmig. Meta-Akad Zwischenbericht. Technical report, Universität Regensburg, Universität Kaiserslautern, Kaiserslautern, 2002.

[Gei00] Dr. Friedrich Geißelmann. Meta-Akad, Metadatenzugang für akademisches Lehr und Lernmaterial, Forschungsantrag. Technical report, Universität Regensburg, Universität Kaiserslautern, Kaiserslautern, Regensburg, 2000.

[Geo02] Amanda G. Watlington Ph.D. Georg Smith. Searcher Behavior Shows Top Listings Are Most Important, Search Engine Branding Survey. Technical report, iProspect Inc., Arlington, MA 02476, 2002.

[Gri98] Frank Griffel. *Componentware, Konzepte und Techniken eines Softwareparadigmas*. dpunkt-Verlag für digitale Technologie GmbH, Heidelberg, 1998.

[Hob01] Dr. Lilian Hobbs. Oracle 9i Materialized Views. Technical report, Universität Regensburg, Universität Kaiserslautern, Redwood Shores, 2001.

[Koz99] Dr. Wojtek Kozaczynski. International Workshop on Component-Based Software Engineering: Composite Nature of Component. Technical report, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA, 1999.

[Mel03] Carnegie Mellon. How Do You Define Software Architecture? http://www.sei.cmu.edu/architecture/definitions.html, 2003.

[Mic01] Sun Microsystems. Java 2 Enterprise Edition Specification, v1.3. http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf, 2001.

[Mic03a] Microsoft. Die Microsoft .NET-Plattform. http://www.microsoft.com/germany/themen/net/plattform/index.htm, 2003.

[Mic03b] Sun Microsystems. Java 2 Enterprise Edition Home Page. http://java.sun.com/j2ee/, 2003.

[Mic03c] Sun Microsystems. Java BluePrints, Fast Lane Reader. http://java.sun.com/blueprints/patterns/FastLaneReader.html, 2003.

[Mic03d] Sun Microsystems. Java Web Start. http://java.sun.com/products/javawebstart/, 2003.

[Mic03e] Sun Microsystems. Scaling the N-Tier Architecture. http://wwws.sun.com/software/whitepapers/wp-ntier/wp-ntier.pdf, 2003.

[Mic03f] Sun Microsystems. The Java Network Launching Protocol (JNLP) and Java Web Start. http://developer.java.sun.com/developer/technicalArticles/Programming/jnlp/, 2003.

[Moh02] C. Mohan(u.a.). Middle-tier Database Caching for e-Business. Technical report, IBM Almaden Research Center, San Jose, CA 95120, USA, 2002.

[Nag02] Nagraj Alur, Peter Haas, Daniela Momiroska, Paul Read, Nicholas Summers, Virginia Totanes, Calisto Zuzarte. *DB2 UDBs High Function Business Intelligence in e-business.* IBM Redbooks, San Jose, 2002.

[Rom99] Ed Roman. *Mastering Enterprise Java Beans and the Java 2 Platform, Enterprise Edition.* Wiley Computer Publishing, New York, 1999.

[Sch99] Doug Schmidt. How to Make Software Reuse Work for You. Technical report, C++ Report, 1999.

[SSJtET02] Inderjeet Singh, Beth Stearms, Mark Johnson, and the Enterprise Team. *Designing Enterprise Applications with the J2EE Platform, Second Edition.* Addison-Wesley, Boston, 2002.

[Szy98] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, Harlow (u.a.), 1998.

[Web03] Christian Weber. Realisierung einer automatischen Klassifizierungs- und Beschlagwortungskomponente zur Nutzung im Projekt META-AKAD. Technical report, Universität Kaiserslautern, Kaiserslautern, 2003.

[Wik03] Wiki. Component definition. http://www.c2.com/cgi/wiki?ComponentDefinition, 2003.

[WR00] Russel Winder and Graham Roberts. *Developing Java Software*. John Wiley & Sons LTD, Chichester, New York (u.a.), 2000.