Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik IV
Prof. Dr. rer. nat. O. Spaniol

Diplomarbeit

Optimierung der Dienstauswahl in einem CORBA-Trader unter dem Aspekt dynamischer Lastverteilung

(Optimizing the Set of Selected Services in a CORBA Trader by Integrating Dynamic Load Balancing)

vorgelegt von:
Cand. Inf. Helmut Neukirchen
Matr.-Nr. XXXXXX

März 1999

Betreuer:
Prof. Dr. rer. nat. O. Spaniol
Dipl. Inf. D. Thißen

Zweitgutachter:
Prof. Dr. Ing. M. Nagl

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Aachen, den 9.3.1999

Helmut Neukirchen

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Wunsch nach immer leistungsfähigeren Computeranwendungen und die Möglichkeit, über das Internet weltweit angebotene Dienste zu nutzen, haben zu einem beachtlichen Wandel der Softwaresysteme geführt. Die vorherrschenden Einzelplatzanwendungen werden in diesem Zuge von verteilten Anwendungen abgelöst. Sie versprechen eine effiziente Nutzung der vorhandenen Ressourcen und unterstützen den Trend der Arbeitswelt hin zur flexiblen, computergestützten Gruppenarbeit.
Im Gegensatz zu einem monolitischen, zentralen System sind Verteilte Systeme in der Lage, die angesprochenen Anforderungen zu erfüllen, da sie Aufgaben auf mehrere Rechnerknoten verteilen können und so eine Steigerung der Leistungsfähigkeit gegenüber einem einzelnen System ermöglichen.
Gleichzeitig mit dem Aufkommen verteilter Technologien wurde jedoch deutlich, daß sich die Softwareentwicklung bei verteilten Anwendungen weitaus komplexer als bei herkömmlichen Systemen gestaltet. Zur Lösung dieses Problems werden standardisierte Verteilungsplattformen benötigt, die in der Lage sind, die Verteilung transparent erscheinen zu lassen. Offene Standards helfen dabei, Heterogenitäten zwischen den einzelnen Komponenten eines Systems zu überbrücken. Ein solcher Standard, der zunehmend an Bedeutung gewinnt, liegt seit einiger Zeit mit CORBA vor.
Einen zentralen Bestandteil des CORBA-Standards bildet die einheitliche Schnittstellenbeschreibung, durch die es möglich ist, Dienste nach dem Baukastenprinzip zu nutzen. Durch Intranetze und das Internet kann so auf ein firmenweites oder gar weltweites Angebot von Diensten zugegriffen werden. Es entsteht ein offener Dienstmarkt, zu dessen Nutzung eine geeignete Vermittlung benötigt wird. Das Trading-Konzept bieten eine derartige Unterstützung, indem Dienste anhand ihrer Dienstleistung typisiert und durch ihre Diensteigenschaften näher charakterisiert werden.
Als weiterer wichtiger Aspekt von Verteilten Systemen ist die mögliche Redundanz von Komponenten anzusehen. Hierdurch kann zum einen die Ausfallsicherheit gesteigert werden. Zum anderen wird das System bezüglich seiner Leistungsfähigkeit skalierbar. Dennoch kann es auch in einem ausreichend dimensionierten Verteilten System zu temporären Leistungsengpässen kommen, wenn eine Ressource intensiv von mehreren Dienstnutzern gleichzeitig genutzt wird. Diese Engpässe können vermindert

werden, indem Dienstnutzungen auf mehrere ähnliche Dienstanbieter verteilt werden. In dieser Arbeit wird ein Konzept vorgestellt, das eine Lastverteilung in einem Dienstmarkt ermöglicht. Hierbei kommt ein Trader zum Einsatz, dessen Ziel es ist, einen optimalen Dienst auszuwählen. Diese Optimalität bezieht sich auf zwei Aspekte, zwischen denen ein Kompromiß zu schließen ist. Einerseits ist dies die Dienstbearbeitungsdauer, die neben der Leistungsfähigkeit eines Rechnerknotens auch von der aktuellen Last abhängig ist. Andererseits soll weiterhin die Güte der Eigenschaften eines Dienstangebots berücksichtigt werden, um das Potential, das sich durch das Tradingkonzept ergibt, nutzen zu können. Von einer transparenten Integration einer derartigen Lastverteilung in die Dienstvermittlung können alle Dienstnutzer profitieren. Im Rahmen dieser Arbeit wurde ein entsprechendes Konzept unter der CORBA-Plattform Orbix implementiert und getestet.

Die vorliegende Arbeit gliedert sich in sechs Kapitel. Zunächst wird auf die zum weiteren Verständnis von Dienstvermittlung und Lastverteilung benötigten Grundlagen von Verteilten Systemen eingegangen. Neben Dienstvermittlungskonzepten und Lastverteilungsstrategien kommen dabei auch das Management von Verteilten System, Kommunikationsprinzipien sowie der CORBA-Standard zur Sprache.

Im dritten Kapitel werden existierende Arbeiten im Umfeld von Dienstvermittlung und Lastverteilung vorgestellt. Ein Schwerpunkt wird dabei auf die Integration dieser beiden Aspekte gelegt. Aus den Schwächen der betrachteten Arbeiten ergeben sich Anforderungen an einen verbesserten Entwurf. Diese Verbesserungsvorschläge bilden zusammen mit weiteren Entwurfsentscheidungen den Schluß dieses Kapitels. Sie geben zusammen mit einer Einordnung des eigenen Ansatzes bezüglich der bestehenden Systeme einen Ausblick auf das zu entwickelnde Konzept zur Optimierung der Dienstauswahl unter dem Aspekt dynamischer Lastverteilung.

Das aus der Anforderungsspezifikation entwickelte Objekt-Modell wird in der ersten Hälfte von Kapitel 4 beschrieben. Die zweite Hälfte enthält einige ausgewählte Aspekte der Implementierung der realisierten Trader/Load Balancer-Kombination. Neben den strukturellen Eigenschaften wird in diesem Kapitel auch auf das Verhalten der entstandenen Komponenten eingegangen.

Eine Bewertung der vorgenommenen Implementierung und der verwendeten Algorithmen wird im fünften Kapitel vorgenommen. Dies geschieht anhand von Leistungsmessungen in künstlich generierten Testszenarien. Es erfolgt ein Vergleich der unterschiedlichen implementierten Strategien sowohl untereinander als auch bezüglich eines herkömmlichen Traders.

Kapitel 6 bildet den Abschluß und faßt die in dieser Arbeit gewonnen Resultate zusammen. Anhand der offen gebliebenen Fragen wird ein Ausblick auf weitere, verfolgenswert erscheinende Ansätze gegeben.

Kapitel 2

Dienstvermittlung und Lastbalancierung in Verteilten Systemen

Im folgenden soll ein Überblick über Verteilte Systeme und die dort eingesetzten Konzepte gegeben werden. Neben den Grundlagen, die im wesentlichen auf [Tan95, Pop96] basieren, wird dabei detaillierter auf die im weiteren Verlauf dieser Arbeit behandelten Schwerpunkte Dienstvermittlung und Lastverteilung eingegangen.

2.1 Verteilte Systeme

Laut [SPM94] handelt es sich bei einem Verteilten System um ein *„System mit räumlich verteilten Komponenten, die keinen gemeinsamen Speicher benutzen und einer dezentralen Administration unterstellt sind. Zur Ausführung gemeinsamer Ziele ist eine Kooperation der Komponenten möglich. Werden von den Komponenten Dienste angeboten oder genutzt, so entsteht ein Client/Server-System, im Falle einer zusätzlichen zentralen Dienstvermittlung ein Tradingsystem.“*[1]
Ein Verteiltes System besteht demnach – wie in Abbildung 2.1 dargestellt – aus mehreren Computern (Rechnerknoten), die untereinander vernetzt sind. Die Softwarekomponenten auf den einzelnen Rechnerknoten werden gleichzeitig ausgeführt und kommunizieren über das Netzwerk untereinander, um Daten auszutauschen.
Gegenüber herkömmlichen, zentralen Systemen bietet der verteilte Ansatz einige Vorteile, die – nicht zuletzt wegen der dadurch möglichen Kosteneinsparung – dazu geführt haben, daß sich verteilte Anwendungen immer stärker durchsetzen. Im gleichen Maße verlieren dabei Großrechner an Bedeutung, da sie bei Aufgaben, die sich genausogut auf mehrere kleinere Rechner verteilen lassen, ungleich teurer in der Anschaffung sind.

[1] Laut [Tan95] definiert Leslie Lamport ein Verteiltes System wie folgt: „*Ein System, mit dem man nicht arbeiten kann, weil ein Rechner ausgefallen ist, von dem man noch nie etwas gehört hat.*“

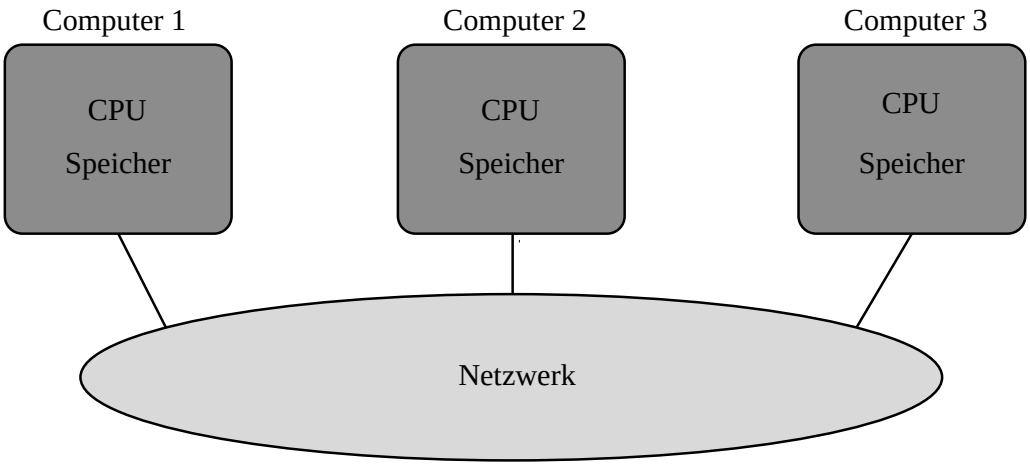

Abbildung 2.1: Ein einfaches Verteiltes System

Zu den Vorteilen von Verteilten Systemen zählen u.a. [ZFD93]:

- Wirtschaftlichkeit: Ein Verteiltes System ist bei gleicher Leistungsfähigkeit preiswerter als ein entsprechender Großrechner.
- Geschwindigkeit: Durch Parallelisierung von Arbeitsschritten können Geschwindigkeiten erreicht werden, die ein einzelner Rechner nie erreichen kann.
- Gemeinsame Nutzung von Betriebsmitteln: Gemeinsam benötigte Daten können von allen genutzt werden; teure Geräte können von allen Komponenten des Systems benutzt werden.
- Verteiltheit: Die z.B. aufgrund von kooperativer Gruppenarbeit notwendige räumliche Verteilung von Rechnersystemen, die miteinander kommunizieren, wird durch Verteilte Systeme erst möglich.
- Fehlertoleranz: Bei Ausfall eines Rechnerknotens kann das Restsystem noch weiterarbeiten und die Funktionalität des ausgefallen Rechners übernehmen.
- Skalierbarkeit: Die Rechenleistung kann in beliebigen Schritten durch Hinzufügen von neuen Rechnerknoten an die benötigte Leistung angepaßt werden.
- Flexibilität: Eine Änderung in der Struktur des Systems ist leicht möglich, um sich so geänderten Gegebenheiten anzupassen.

Damit ein Verteiltes System die genannten Vorteile ausspielen kann, müssen jedoch einige Schwierigkeiten, die sich aufgrund der Verteiltheit ergeben, gelöst werden. Als problembehaftet sind die folgenden Punkte anzusehen:

- Softwareentwicklung: Durch die Verteilung von Daten und Algorithmen ergeben sich neuartige softwaretechnische Probleme. Damit die Erstellungskosten

verteilter Anwendungen im Bereich von zentralen Anwendungen liegen, werden Konzepte zur nahtlosen Integration der Verteilung benötigt.

- Heterogenität: Ein System kann aus Komponenten verschiedener Hersteller bestehen. Zur Überbrückung der herstellerspezifischen Unterschiede werden offene Standards benötigt.

- Datenschutz und Datensicherheit: Durch die Verteilung können schützenswerte Daten nicht mehr so leicht gegen unerlaubte Zugriffe abgeschottet werden. Hierfür müssen Authentifizierungs- und sonstige Sicherheitsmechanismen auf die Konzepte der Verteilten Systeme übertragen werden.

- Fehleranfälligkeit: Je größer ein System wird, desto wahrscheinlicher wird es, daß eine Komponente ausfällt. Um die Auswirkung zu begrenzen, muß es möglich sein, daß andere Komponenten die Aufgaben der ausgefallenen Komponente übernehmen. Zusätzlich kommt in Verteilten Systemen durch die Kommunikation über ein Netzwerk eine zusätzliche, durch entsprechende Übertragungsprotokolle zu berücksichtigende Fehlerquelle im Programmdatenfluß hinzu.

- Synchronisation: Die Synchronisation von Uhren und Abläufen gestaltet sich schwieriger als in zentralen Systemen und erfordert neuartige, verteilte Algorithmen.

- Lokalisierung von Komponenten: Im Gegensatz zu Einzelplatzanwendungen, bei denen sich alle benötigten Programmkomponenten auf dem lokalen Rechner befinden, benötigen verteilte Anwendungen eine entsprechende Unterstützung, da durch den Einsatz in einem offenen Dienstmarkt und aufgrund von Objektmigration erst zur Laufzeit ermittelt werden kann, auf welchem Rechnerknoten sich ein jeweils gesuchtes Objekt befindet.

- Überlast: Der Vorteil der Skalierbarkeit geht schnell verloren, wenn aufgrund von lokalen Engpässen die gesamte Systemleistung herabgesetzt wird. Hierbei muß zwischen einer Überlast des Netzes aufgrund übermäßiger Kommunikationsvorgänge und überlasteten Rechnerknoten durch erhöhtes Berechnungsaufkommen unterschieden werden. Zur Vermeidung solcher Überlastsituationen ist neben der Reduzierung von Flaschenhälsen auf eine gleichmäßige Verteilung der Last auf mehrere Komponenten zu achten.

2.1.1 Das objektorientierte Verteilungsmodell

Die im vorhergehenden Abschnitt aufgeführten Probleme können zum Teil vermieden werden, wenn eine Kontextunabhängigkeit [ZFD93] der Komponenten gegeben ist, d.h. einzelne Komponenten ohne Änderung in unterschiedlichen Umgebungen eingesetzt werden können. Als wichtige Grundanforderungen an ein Paradigma, das die

Entwicklung eines Verteilten Systems unterstützt, ergeben sich dementsprechend u.a. [Red96, Kov94]:

- Verteilungstransparenz: Die Aspekte der Verteilung (Auf welchem Rechner befinden sich die Daten? Wie ist das System strukturiert?) sollen der Anwendung – und somit dem Entwickler – verborgen bleiben, damit wie gewohnt auf Daten und Programmteile zugegriffen werden kann.

- Interoperabilität und Offenheit: Um eine Zusammenarbeit der unterschiedlichen Rechner und Betriebssysteme sowie Netze, aus denen das System bestehen kann, zu ermöglichen, müssen deren Heterogenitäten durch herstellerübergreifende, offene Standards überbrückt werden.

- Flexibilität und Modularität: Eine modulare Softwareentwicklung ermöglicht die effiziente Wiederverwendung von existierenden Komponenten. Die Module bilden dabei die Einheiten, in die das Gesamtsystem zerlegt und auf einzelne Rechnerknoten verteilt werden kann. Durch Änderung an wenigen Modulen kann das System effizient und flexibel neuen Gegebenheiten angepaßt werden.

Es hat sich gezeigt, daß ein objektorientierter Ansatz [RBP+93, Mey90] bisher am besten geeignet ist, diese Anforderungen zu erfüllen [Sch91]: Die objektbasierte Sichtweise sorgt durch Abstraktion dafür, daß Details verborgen werden [Nag90], wodurch die Forderungen nach Verteilungstransparenz und Interoperabilität erfüllt werden. Da bei der objektorientierten Herangehensweise ein Gesamtsystem aus einer Ansammlung von einzelnen Objekten gebildet wird, ergibt sich automatisch ein modularer Aufbau, der darüber hinaus durch die übrigen Aspekte des objektorientierten Ansatzes noch an weiterer Flexibilität gewinnt.
Beim objektorientierten Paradigma wird als grundlegendes Konstrukt das Objekt verwendet; es enthält Datenstrukturen und Algorithmen zugleich und ist somit ideal für die Verteilung sowohl von Daten als auch Abläufen geeignet. Die in der realen Welt anzutreffenden Objekte einer Anwendung spiegeln sich bei der Modellierung dieser Anwendung im Computer in Form von solchen Softwareobjekten wieder. Da der Aufruf der einzelnen Operationen eines Objekts konzeptionell bereits eine gewisse Kommunikation erfordert, um das passende Objekt oder eine Operation (sog. Methode) zu finden (dynamisches Binden), gestaltet sich der Übergang zu einem Verteilten System relativ einfach.
Einzelne Objekte lassen sich auf verschiedenen Rechnerknoten instantiieren und stellen ihre Dienste über Schnittstellen nach außen anderen Objekten zur Verfügung. Eine verteilte Anwendung setzt sich dann aus den Instanzen solcher Objekte zusammen, die gegenseitig Operationen aufrufen. Die Kommunikation erfolgt dabei wie in Abbildung 2.2 über ihre Schnittstellen.
Zum Aufruf solcher Operationen wird ein entfernter Methodenaufruf verwendet, der das Konzept des entfernten Unterprogrammaufrufs (remote procedure call, RPC) in die objektorientierte Welt überträgt [Sch91].

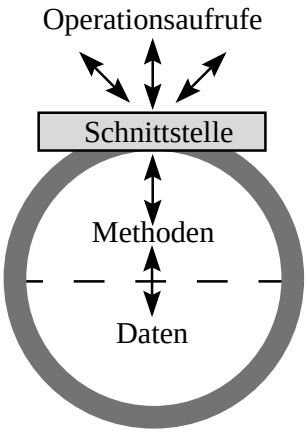

Abbildung 2.2: Aufbau eines Objekts

Frühe Verteilte Systeme basierten auf dem Client-Server-Modell, bei dem über Ein-/Ausgabeprimitive („send“ und „receive“) Daten von Benutzerprozessen (Clients) an Dienstprozesse (Server) geschickt, dort verarbeitet und zur Ergebnisübermittlung wieder zurückgesendet wurden. Das darauffolgende Paradigma greift den herkömmlichen Unterprogrammaufruf auf und erreicht dadurch die geforderte Verteilungstransparenz. Die Idee des entfernten Unterprogrammaufrufs sieht vor, den Aufruf von Diensten, die sich auf einem anderen Rechner befinden, wie einen herkömmlichen Unterprogrammaufruf erscheinen zu lassen. Der Aufruf von entfernten Diensten unterscheidet sich für den Programmierer nicht von dem Aufruf eines Unterprogramms, das denselben Dienst lokal erbringt.
Für das objektorientierte Verteilungsmodell muß dieses Konzept noch erweitert werden. Neben der nun möglichen Objektvererbung und Operationspolymorphie [Mey90, RBP+93], die einer zusätzlichen Beachtung bedürfen, können durch die Verwendung eines objektorientierten Ansatzes nun nicht nur Unterprogramme, sondern auch Daten transparent verteilt werden. Dadurch wird es u.a. möglich, Referenzen auf entfernte Daten bzw. Objekte zu verwenden, weshalb die Behandlung von Zeigern ebenfalls an die verteilte Welt angepaßt werden muß [Sch91].
Die beschriebenen Vorteile und die dafür zu lösenden Probleme machen deutlich, daß Verteilte Systeme unter dem Aspekt der Softwareentwicklung zugleich ein Problem und eine Chance darstellen: Zum einen werden durch die Verteilung völlig neuartige Probleme aufgeworfen, zum anderen rückt die Vision der Wiederverwendbarkeit von Software nach dem Baukastenprinzip [Nag90] einen Schritt näher, da in Verteilten Systemen eine Kontextunabhängigkeit gegeben ist [Sta95, APRA98, Sch91].

2.1.2 Kommunikation in Verteilten Systemen

Eine elementare Grundlage für jedes Verteilte System ist eine funktionierende Kommunikationsinfrastruktur. Um die in den vorhergehendenen Abschnitten geforderte

Offenheit des Systems zu erreichen, werden offene Kommunikationsarchitekturen benötigt. Als Modell, das einen Rahmen dafür bietet, hat sich das Open Systems Interconnection-Referenzmodell (OSI-RM) [ISO84] der internationalen Standardisierungsbehörde ISO bewährt.
Um eine offene Kommunikation zu gewährleisten, sieht das OSI-Modell vor, die jeweils spezifischen Aspekte der Kommunikation in insgesamt sieben aufeinanderaufsetzende Schichten mit definierten Schnittstellen zu kapseln. Die in den einzelnen Ebenen behandelten Aspekte reichen von der Übertragungshardware über die Sicherstellung eines fehlerfreien Transports bis zur eigentlichen Anwendung. Die genaue Schichtung ist in Abbildung 2.3 (a) dargestellt.

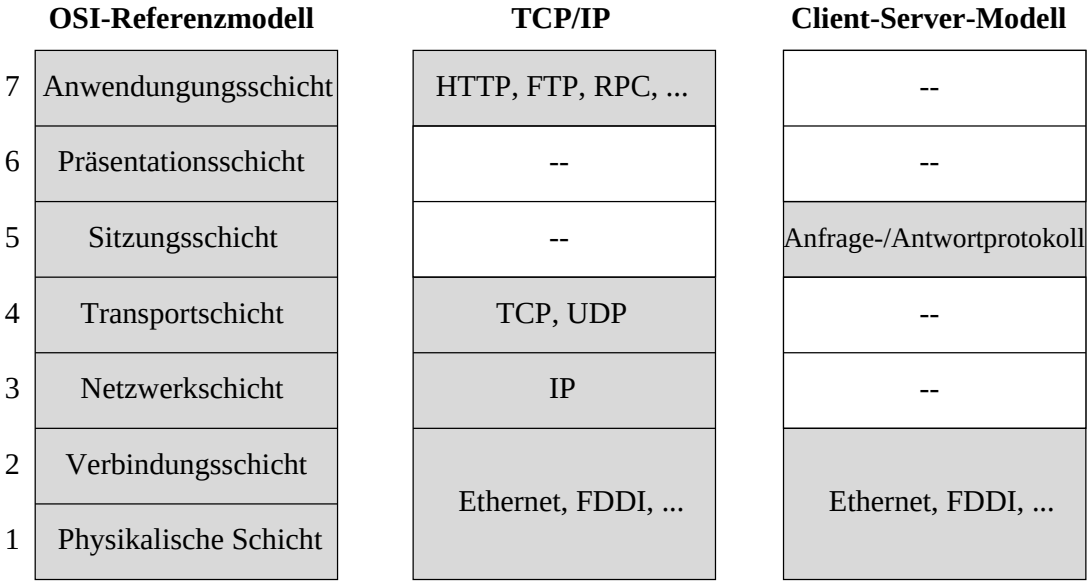

Abbildung 2.3: Die OSI-Schichten, die TCP/IP-Protokollfamilie und das Client-Server-Modell

Von der ISO werden auch konkrete Protokolle zur Implementierung der von den einzelnen OSI-Schichten bereitzustellenden Dienste vorgeschlagen, dennoch hat sich die unabhängig davon entwickelte Internetprotokollfamilie weltweit durchgesetzt [Tan96].
Die Internetprotokollfamilie besitzt zwar kein so sauberes Modell wie OSI, ist aber aufgrund des ebenfalls vorhandenen Schichtenkonzepts und seiner weiten Verbreitung gleichermaßen zur Kommunikation in heterogenen Systemen geeignet. Eine Einordnung der einzelnen Internetprotokolle in das OSI-Referenzmodell ist in Abbildung 2.3 (b) gegeben.
Es lassen sich grundsätzlich verbindungsorientierte und verbindungslose Übertragungsprotokolle unterscheiden. Während das verbindungsorientierte Transmission Control Protocol (TCP) eine Verbindung zwischen Sender und Empfänger aufbaut und eine verlustfreie Informationsübertragung garantiert, fehlt dem verbindungslosen

User Datagram Protocol (UDP) eine derartige Sicherung. Da UDP unmittelbar auf dem recht einfachen Internetprotokoll (IP) der darunterliegenden Schicht aufbaut, ist die Übertragung mittels UDP schneller, aber auch fehlerbehafteter als die per TCP. Die Pakete, die das Internetprotokoll zur Übertragung verwendet, können verlorengehen oder in unterschiedlicher Reihenfolge eintreffen, weshalb TCP zusätzlich eine aufwendige Sicherung und Sortierung der IP-Pakete vornehmen muß [Tan96].
Ein Grund, der – neben der früheren Verfügbarkeit – zur Dominanz der Internetprotokolle beigetragen hat, ist deren – gegenüber OSI-Protokollimplementierungen – i.d.R. höhere Geschwindigkeit. Hierfür ist die effizientere Implementierbarkeit bzw. die geringere Anzahl von Schichten verantwortlich, da jede durchlaufene Schicht einen zusätzlichen Protokolloverhead zur Folge hat. Im Vergleich dazu ist in Abbildung 2.3 (c) das Client-Server-Modell zur Prozeßkommunikation zu sehen, das mit seinem einfachen Anfrage-/Antwortprotokoll direkt auf der Netzwerkhardware aufsetzt und entsprechend schnell arbeiten kann [Tan96].
Die beiden beschriebenen Kommunikationsarchitekturen erlauben eine Kommunikation zwischen einzelnen Rechnerknoten; sie beinhalten aber keine Konzepte zur Kommunikation zwischen einzelnen entfernten Prozessen. Dies wird nebem dem Client-Server-Modell, das sich wegen mangelnder Transparenz nicht durchsetzen konnte, vom bereits im vorherigen Kapitel angesprochenen entfernten Methoden- bzw. Prozeduraufruf geleistet. Dieses Konzept ist in den obersten drei Schichten des OSI-Referenzmodells angesiedelt und setzt auf den übrigen unteren vier Schichten auf. Vom Aspekt der Datenkommunikation her ist es dabei unerheblich, ob ein prozedurales oder ein objektorientes Verteilungsmodell zum Einsatz kommt, weshalb im folgenden vereinheitlichend der Begriff RPC verwendet wird.
Da herkömmliche Programmiersprachen Unterprogramme lediglich lokal aufrufen können, wird eine Erweiterung benötigt, die es dem Benutzerprozeß ermöglicht, den Unterprogrammaufruf über das Netzwerk an den gewünschten Server-Prozeß weiterzuleiten. Dies wird vom sogenannten Client-Stub geleistet. Dieser verpackt die Übergabeparameter des Unterprogrammaufrufs in ein Nachrichtenpacket, das von der Kommunikationsinfrastruktur an den Zielrechner weitergeleitet wird. Dort nimmt der Server-Stub dieses Paket entgegen, packt die übergebenen Parameter aus und ruft damit das gewünschte Unterprogramm auf, um im Anschluß die Rückgabeparameter zurückzusenden. Dieser Ablauf wird in Abbildung 2.4 dargestellt. Neben den beschriebenen Aufgaben muß sich der Stub auch darum kümmern, daß die verwendeten Datenformate der jeweiligen Rechnerknoten kompatibel sind, indem die zu übertragenden Werte in eine rechnerunabhängige Darstellung umgewandelt werden (Marshalling).
Analog zur Semantik des herkömmlichen Unterprogrammaufrufs wird der aufrufende Prozeß blockiert, bis die Rückgabeparameter eingetroffen sind. Da aus Geschwindigkeitsgründen für den RPC oft ein verbindungsloses Übertragungsprotokoll verwendet wird, kann es jedoch vorkommen, daß RPC-Nachrichten verlorengehen. Der Stub muß sich deshalb nach einem Timeout darum kümmern, die Nachricht ein weiteres Mal zu senden oder einen Fehler zu melden. Hierin unterscheidet sich die Semantik des entfernten vom lokalen Unterprogrammauf, bei dem kein Fehler auftreten kann.

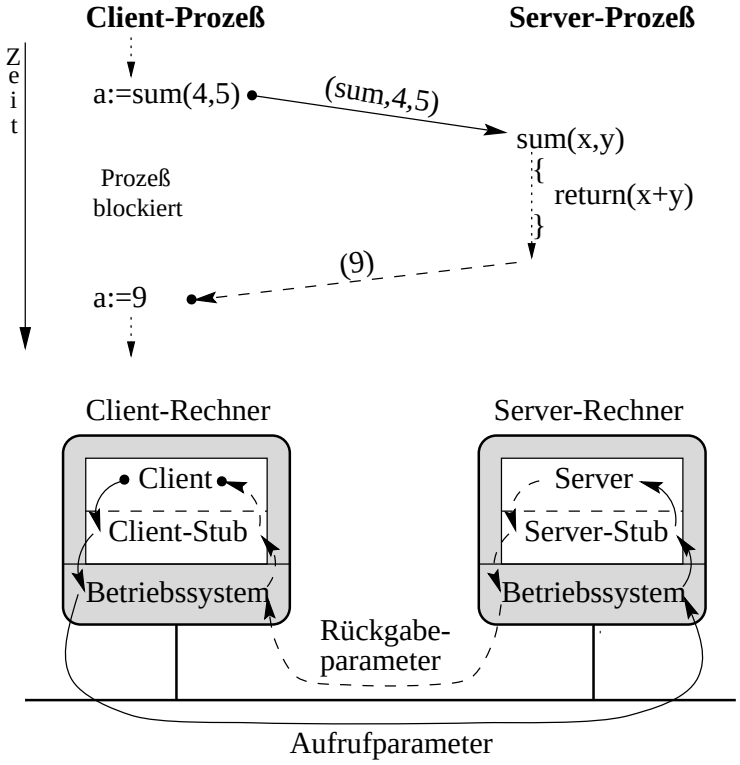

Abbildung 2.4: Schema eines entfernten Prozeduraufrufs

In objektorientierten Sprachen sind üblicherweise programmeigene Routinen zur Ausnahmebehandlung vorgesehen, über die der Aufrufer geeignet auf stub-generierte Fehler reagieren kann.
Die vom Client-Stub generierte RPC-Nachricht ist an ein konkretes aufzurufendes Unterprogramm gerichtet. Das Verteilte System steht dabei vor dem Problem, diese Nachricht einem bestimmten Prozeß auf einem bestimmten Rechnerknoten zuordnen zu müssen. Hierfür wird ein dynamischer Binder benötigt, der aus den Angaben, mit denen sich ein Serverprozeß beim Binder registrieren muß, die passende Zuordnung treffen kann. Abschnitt 2.2 beschäftigt sich eingehend mit den verschiedenen Konzepten, die dazu verwendet werden können.
Der entfernte Unterprogrammaufruf gliedert sich zwar hervorragend in die Welt der bekannten Programmierparadigmen ein, für manche Anwendungsgebiete, die in Verteilten Systemen auftreten, ist dieses Konzept dennoch ungeeignet. Für eine asynchrone Kommunikation, die es den beteiligten Prozessen ermöglicht, weiterzuarbeiten, ohne auf die Antwort des anderen Prozesses warten zu müssen, sind beispielsweise nicht-blockierende Kommunikationsmethoden nötig. Auch Meldungen, die an alle oder mehrere Prozesse gehen sollen (Broad- bzw. Multicasting), lassen sich nicht effizient mit der Zwei-Parteien-Kommunikation des RPC realisieren. Hierfür sind in den meisten Verteilten Systemen gesonderte Konzepte vorgesehen, auf die an dieser Stelle allerdings nicht näher eingegangen werden soll.

2.1.3 Management von Verteilten Systemen

Um die vielfältigen Ressourcen eines Verteilten Systems zu verwalten, zu überwachen und zu koordinieren, wird ein spezielles Management benötigt. Ziel eines derartigen Managements von Verteilten Systemen ist ein möglichst effektiver Einsatz des Gesamtsystems. Eine ausführliche Beschreibung dieses Gebiets wird in [HA93, Slo94, Sei94, LR96, Kau92] gegeben.
Die hier betrachteten Managementaufgaben lassen sich durch Ebenen und funktionale Bereiche strukturieren. Zunächst lassen sich die einzelnen Ressourcen eines Verteilten System der Netzwerkebene, der (Betriebs-)Systemebene und der Anwendungsebene zuordnen. Für das in dieser Arbeit behandelte Gebiet des Anwendungsmanagement bestehen die verwalteten Ressourcen beispielsweise aus den statischen Softwaremodulen und den laufenden Prozessen, sowie den von diesen verarbeiteten Daten.
Als weitere Strukturierungsdimension dient die Aufteilung in unterschiedliche funktionale Managementbereiche. Diese ergeben sich aus den für ein umfassendes Management benötigten grundsätzlichen Aufgaben. Im einzelnen ergeben sich fünf Aufgabenbereiche [ISO89]:

- Fehlermanagement: Es ist für die ordnungsgemäße Funktion eines Systems äußerst wichtig, Fehler im System zu erkennen und zu beheben. Das Fehlermanagement muß entsprechende Funktionen zur Verfügung stellen.

- Konfigurationsmanagement: Dieser Bereich befaßt sich mit der Struktur des Systems und der Konfiguration der einzelnen Komponenten. Neben dem statischen Systemzustand, der vor Inbetriebnahme eingestellt wird, sollte sich dieser Zustand auch dynamisch ändern lassen.

- Abbrechnungsmanagement: Vor allem im kommerziellen Bereich spielt die Erfassung der durch die Nutzung der verwalteten Ressourcen bzw. Dienstleistungen aufgetretenen Kosten eine wichtige Rolle.

- Leistungsmanagement: Überlastsituationen sollen erkannt und durch Lastausgleich abgefangen werden. Hierzu ist es nötig, Statistiken über die Auslastung zu führen, um geeignete Gegenmaßnahmen planen zu können.

- Sicherheitsmanagement: Um Datensicherheit und Datenschutz zu gewährleisten, müssen Zugriffsrechte verwaltet und geregelt werden, anhand derer Authentifizierungsmechanismen den Zugriff auf die jeweiligen Ressourcen zulassen oder auch Alarm auslösen.

Durch Kombination dieser Bereiche lassen sich weitere Aufgabengebiete behandeln. Hierunter fällt z.B. das Dienstmanagement, bei dem die möglichen Dienste, die von einer Anwendung in Anspruch genommen werden können, verwaltet werden. Neben der korrekten Zuteilung der unterschiedlichen Dienstleistungen an die Dienstnutzer ist dabei die Berücksichtigung unterschiedlicher Dienstgüte (Quality of Service, QoS) von

Bedeutung. In Echtzeitsystemen und im Multimediabereich ist die Qualität der Übertragung äußerst wichtig.
Von verschiedenen Seiten wurden Standards entwickelt, welche die Vereinheitlichung, die bei den Kommmunikationsnetzen stattgefunden hat, auch auf deren Management übertragen sollen. Von der Netzwerkebene kommend gibt es dementsprechend wieder einen ISO OSI-Standard (Common Management Information Service bzw. Protocol, CMIS/CMIP) [ISO89] sowie einen Internetstandard (Simple Network Management Protocol, SNMP) [CFSD90], die sich durchgesetzt haben.
Daneben existieren im Bereich des Managements für verteilte Anwendungen zur Zeit noch viele unterschiedliche Systeme, da die Standards der Netzwerkebene sich nicht umfassend hierauf übertragen lassen. SNMP bietet kein objektorientiertes Konzept und aufgrund der angestrebten Einfachheit auch nur eine eingeschränkte Funktionalität; CMIP hingegen erfüllt zwar diese Anforderungen, ist jedoch viel zu komplex und entsprechend aufwendig zu implementieren. Zur Zeit dominieren daher noch proprietäre Lösungen, die sich am OSI-Standard orientieren. Neben dem OSI-Management Framework hat die ISO später speziell für Verteilte Systeme das Open Distributed Processing-Referenzmodell (ODP-RM) [Ray94, ISO95a] standardisiert, das seit neuerer Zeit auch im Managementbereich an Einfluß gewinnt [KN97].
Das ODP-Referenzmodell ist in Schicht 7 des OSI-Referenzmodells anzusiedeln und stellt ein Rahmenwerk für Verteilte Systeme dar. Die darin vorgesehenen Sichten und Modellierungen bieten eine Basis, um offen Verteilte Systeme zu entwickeln. Anhand der zugehörigen Funktionen [ISO95b] können dementsprechend offene Managementarchitekturen erstellt werden [KN97].
Ähnlich wie bei den Verteilungsparadigmen hat sich im Managementbereich – abgesehen von SNMP – ein objektorientiertes Paradigma durchgesetzt. Zentraler Begriff ist hierbei das Managed Object (MO), das eine Managementsicht auf die zu verwaltenden Ressourcen modelliert. Im Gegensatz zur softwaretechnischen Sicht werden durch Managed Objects nicht die datenverarbeitenden Bausteine, sondern die für das Management identifizierten Ressourcen abstrahiert. Deren Status kann über eine eigene Managementschnittstelle, die von der reinen Datenverarbeitungsschnittstelle des entsprechenden Softwareobjekts unabhängig ist, von außen durch einen Manager abgefragt und verändert werden [Kov94, BU96, ISO89].
Die softwaretechnischen Objekte, die die jeweils lokalen Managed Objects eines Rechnerknotens verwalten, werden i.d.R. als (Management-)Agenten bezeichnet. Auf Veranlassung eines externen Managers nehmen die Agenten die eigentlichen Änderungen an den verwalteten Ressourcen vor bzw. leiten Nachrichten (Notifikationen), die ein Managed Object aufgrund eines Ereignisses erzeugt, an einen Manager weiter. Gegenüber einem Manager tritt ein Agent dabei in die Rolle eines Managed Objects, gegenüber einem Managed Object in die Rolle eines Managers. Das Zusammenspiel in einer derartigen Managementarchitektur ist in Abbildung 2.5 dargestellt [Kov94, CPRV96, IDSM93].
Die in einem System verfügbaren Managementinformationen, die sich aus der Menge der verwalteten Managed Objects ergeben, werden als Management Information Base

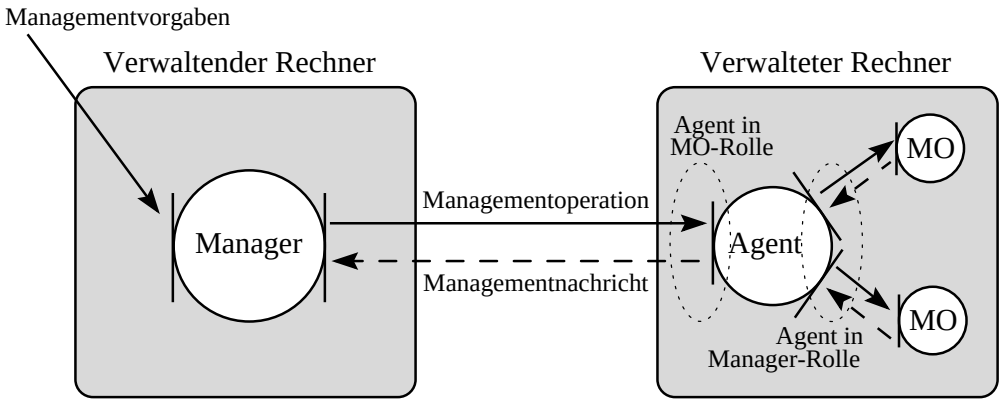

Abbildung 2.5: Prinzipieller Aufbau einer verteilten Managementarchitektur

(MIB) bezeichnet. Das OSI-Modell läßt offen, in welcher Form sich die MIB in einer Implementierung wiederfindet [CPRV96, ISO89].

2.1.4 Monitoring von Verteilten Systemen

Da die Management Information Base auch veränderliche Informationen beinhalten kann, wird für die Erfassung solcher Daten eine spezielle Unterstützung benötigt. Dies wird vom Monitoring geleistet.
Monitoring umfaßt alle Vorgänge, die nötig sind, um dynamische Informationen in einem Verteilten System zu sammeln [MS93]. Weil sich Managemententscheidungen nach derartigen Informationen richten, bildet die Erfassung und Aufbereitung von Systeminformationen eine bedeutende Grundlage für das Management [IDSM93]. Die Rolle, die das Monitoring im Managementvorgang einnimmt, wird in Abbildung 2.6 deutlich. Ein guter Überblick über das Monitoring von Verteilten Systemen wird in [MS93, JLS+87] gegeben.

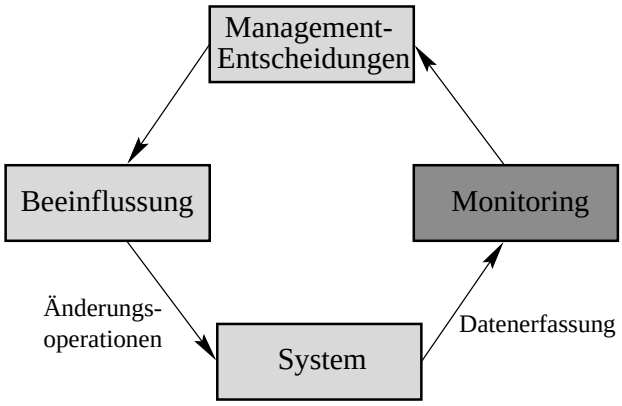

Abbildung 2.6: Monitoring im Kontext des Managements

Neben Anwendungen, die die gesammelten Informationen aufbereiten und darstellen, werden Mechanismen benötigt, um die gewünschten Informationen am Ort des Auftretens zu erfassen und anschließend weiterzuverbreiten bzw. für einen späteren Abruf zu speichern. Dies wird von Monitorobjekten geleistet, die – ähnlich einem Agenten – von den zu überwachenden Ressourcen Daten über deren Zustand ermitteln und weiterverarbeiten.
Neben einer zeitbasierten Meßwertaufnahme in regelmäßigen Abständen ist auch ein ereignisbasiertes Monitoring möglich, bei dem Ereignisse, die eine Zustandsänderung der überwachten Ressource signalisieren, erkannt und weitergemeldet werden müssen. In diesem Zusammenhang tritt das Problem der zeitlichen korrekten Einsortierung der Daten auf, da in einem Verteilten System die Reihenfolge des Eintreffens von Informationen nicht unbedingt mit der Reihenfolge der Entstehung übereinstimmen muß und Zeitmarken aufgrund unsynchronisierter Uhren nicht eindeutig sind. Damit die von verschiedenen Monitoren erfaßten Daten gemeinsam genutzt werden können, muß der Monitor darüber hinaus die Werte über Metriken in vergleichbare Werte umrechnen und in einem einheitlichen Datenformat zur Verfügung stellen.
Bei weiter Auslegung des Managementbegriffs fallen alle per Monitoring gesammelten Daten in den Managementbereich. Dementsprechend lassen sich auch die dort identifizierten fünf Bereiche wie beispielsweise die Fehlerbehebung bzw. -suche oder etwa die Leistung des Systems auch zur Klassifikation der Einsatzgebiete des Monitoring benutzen. Diese Bereiche können wiederum dem Ursprung der Daten entsprechend in Netzwerk-, System- und Anwendungsebene unterteilt werden.
Der Bereich des Leistungsmonitoring soll im folgenden beispielhaft vorgestellt werden, da er im weiteren Verlauf der vorliegenden Arbeit eine wichtige Rolle spielt. Um die Last innerhalb einer verteilten Anwendung zu bestimmen, wird ein Monitor benötigt, der die lastspezifischen Merkmale des zu überwachenden Objekts mißt. Dazu bietet es sich beispielsweise an, die vom Objekt erzeugte CPU-Last oder die Antwortzeiten auf die Anfragen der Clients zu messen. Während für die erste Variante ein Betriebssystemaufruf auf dem betreffenden Rechnerknoten reicht, muß für die zweite ein sogenannter Sensor in das zu überwachende Softwareobjekt eingebaut werden, um dort die entsprechenden Zeiten zu messen und über eine Monitoring-Schnittstelle an den Monitor zu übertragen. Bei der Verwendung von Sensoren muß das objektbasierte Konzept des Versteckens von internen Informationen durchbrochen werden, da ja gerade diese internen Daten eines Objekts erfaßt und nach außen übermittelt werden sollen [CPRV96].
Als zusätzliches Problem ist besonders beim Leistungsmonitoring zu beachten, daß das Monitoring selber wiederum Einfluß auf das System hat, da die Übertragung der Meßdaten Netzwerklast erzeugt und der Monitor selber Rechenzeit und Speicher benötigt, da es sich beim Monitor um ein herkömmliches Softwareobjekt handelt. Daher ist es notwendig, einen Kompromiß zwischen den gegensätzlichen Zielen einer möglichst hohen Genauigkeit bzw. Aktualität der Meßdaten und einer möglichst geringen zusätzlich erzeugten Last zu finden. Hardware-Monitore würden diesen Effekt reduzieren, sie sind allerdings auch teurer und unflexibler, weshalb sie selten eingesetzt werden.

2.1.5 Verteilungsplattformen

Die in den vorhergehenden Abschnitten vorgestellten Konzepte sollen dazu dienen, der Anwendung bzw. dem Anwendungsprogrammierer die Aspekte der Verteilung zu verbergen bzw. den Umgang mit dem System zu vereinfachen. Da z.Z. keines der verbreiteten Betriebssysteme die dafür notwendigen Eigenschaften zur Verfügung stellt, wird hierfür zusätzliche Software benötigt. Diese Lücke zwischen Anwendung und Betriebssystem wird von den Verteilungsplattformen geschlossen. Hierfür ist – ihrer in Abbildung 2.7 gezeigten Lage entsprechend – der Begriff Middleware üblich [Pop96].

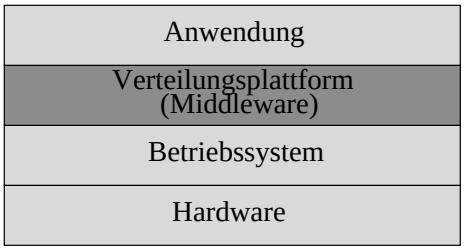

Abbildung 2.7: Einordnung der Middelware in eine grobe Systemarchitektur

Verteilungsplattformen stellen die benötigte Laufzeitumgebung (u.a. den dynamischen Binder) und die Programmiersprachenerweiterung (Client- und Serverstub) bereit. Zu diesem Zweck wird eine Interface Definition Language (IDL) benutzt, mit der die Schnittstellen der verteilten Komponenten definiert werden. Damit die derart definierten Operationen in den herkömmlichen Programmiersprachen aufgerufen werden können, erzeugt ein IDL-Compiler aus diesen Definitionen Include-Dateien und Programmcode, die in das herkömmliche Anwendungsprogramm eingebunden werden und den jeweiligen Stub enthalten. Der dabei übliche grundsätzliche Ablauf zur Erzeugung einer ausführbaren Datei ist der Abbildung 2.8 zu entnehmen [Pop96, Red96].

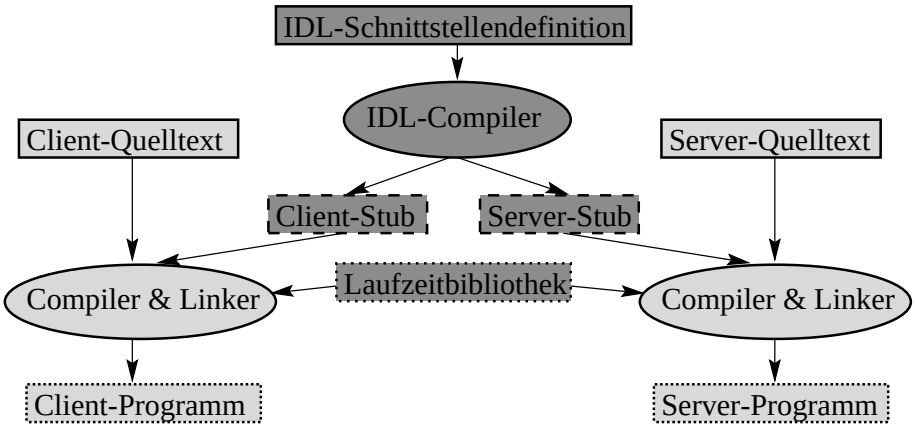

Abbildung 2.8: Programmerzeugung mit einer Verteilungsplattform

Der bereits in Abschnitt 2.1.3 angesprochene ODP-Standard stellt lediglich ein Referenzmodell für Verteilungsplattformen dar, indem er als Rahmenwerk beschreibt, welche Voraussetzungen gegeben sein müssen bzw. welche Herangehensweisen eingesetzt werden sollen, um ein offenes Verteiltes System zu konstruieren. Eine konkrete Implementierung ist durch ODP allerdings nicht vorgegeben [Ray94, ISO95a]. Dementsprechend existieren verschiedene Ansätze für Verteilungsplattformen, die mehr oder weniger konform zum ODP-Referenzmodell sind. Zu den wichtigsten zählen Distributed Computing Environment (DCE) [OSF94] der Open Software Foundation (OSF), ActiveX/Distributed Component Object Model (DCOM) [BK96] von Microsoft und diverse Implementierungen der Common Object Request Broker Architecture (CORBA) [OMG98] der Object Management Group (OMG).
Während DCE an Bedeutung verliert, weil es das objektorientierte Paradigma nicht unterstützt [YD96], hat Microsoft mit DCOM den DCE RPC um eine objektorientierte Ebene erweitert und versucht so, einen Standard für die Personal Computer-Welt zu setzen, der allerdings nicht sehr offen ist und aufgrund seiner historischen Entwicklung keine saubere Architektur besitzt [CHY+97]. Außerhalb der Microsoft Windows-Systeme setzt sich seit einiger Zeit der CORBA-Standard durch, der als offener Standard von vielen Herstellern unterstützt wird. Da keine Referenzimplementierungen vorgesehen ist, liegen diverse freie und kommerzielle CORBA-Implementierungen für viele verschiedene Hardwareplattformen vor. Hierdurch ist es u.a. möglich, die Heterogenitäten zwischen der UNIX und PC-Welt zu überbrücken [CHY+97, YD96, Sta95, APRA98]. Beispielhaft sei neben dem freien MICO [PR98] und dem weit verbreiteten VisiBroker von Inprise [Inp98] als ebenfalls weit verbreitete CORBA-Implementierung Orbix [Ion97a, Ion97b] von Iona genannt, auf der die vorliegende Arbeit basiert.
Ein recht guter Überblick zum CORBA-Standard der Object Management Group wird in [YD96] gegeben. Die 1989 gegründete OMG ist ein Konsortium von mittlerweile fast tausend Firmen, das zum Ziel hat, verteilte objektorientierte Technologien zu fördern, indem ein offener Standard dafür geschaffen wird [Sta95, APRA98, Pop96].
Den elementarsten Teil der in Abbildung 2.9 dargestellten Object Management Architecture (OMA) [OMG95a], die den OMG-Standard für offene objektorientierte Verteilungsplattformen beschreibt, bildet die Common Object Request Broker Architecture [OMG98]. Wichtigste Bestandteile von CORBA sind die Spezifikation des Object Request Brokers (ORB), der den dynamischen Binder und den entfernten Methodenaufruf realisiert, und die Abbildung der IDL-Schnittstellenbeschreibung in verschiedene gebräuchliche Programmiersprachen (Language-Mappings) [Sta95, Red96].
Darüber hinaus sind in der OMA Object Services standardisiert (CORBAservices [OMG97]). Sie bieten die Grundfunktionen, die für die Arbeit mit verteilten Objekten benötigt werden, an. So steht für ein kontextunabhängiges dynamisches Binden der Object Naming Service zur Verfügung, der es ermöglicht, Objekte anhand eines symbolischen Namens statt über Objektreferenzen zu adressieren. Weitere Funktionen ermöglichen z.B. die Migration von Objekten oder ihre persistente Speicherung auf Datenträgern.

Die Common Facilities, die häufig gebrauchte Anwendungsfunktionen, wie sie etwa für grafische Benutzeroberflächen oder zur Dokumentenverwaltung benötigt werden, bereitstellen, sind zwar standardisiert (CORBAfacilities [OMG95b]), aber bisher nur optional in einigen OMA/CORBA[2]-Implementierungen vorhanden.
Als weiteren Bereich identifiziert die OMA die anwendungsspezifischen Domain Interfaces, die Dienste für unterschiedliche Anwendungsbereiche wie CAD, den Finanzsektor oder etwa den der Telekommunikationsindustrie zusammenfassen.

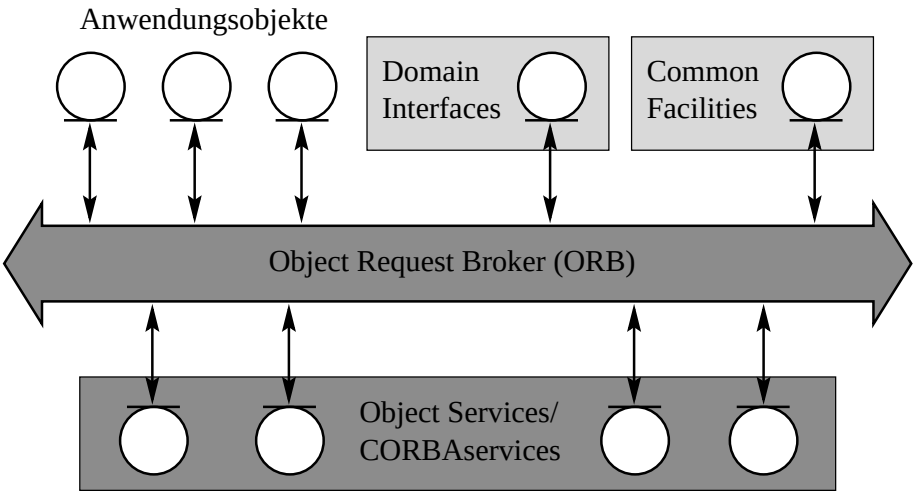

Abbildung 2.9: Die Bestandteile der Object Management Architecture (OMA)

Der ORB bildet die Kommunikationszentrale der OMA. Der von ihm bereitgestellte entfernte Methodenaufruf realisiert allerdings nicht alle Konzepte des objektorientierten Paradigmas [RBP+93]. So ist es nicht erlaubt, per IDL definierte Operatoren polymorph zu gestalten bzw. zu überladen. Die Objektübergabe in Aufrufparametern ist nur eingeschränkt möglich und das Konzept der Objektidentität wurde erst in der Version 2.0 des CORBA-Standards eingeführt. Zwar benutzt der dynamische Binder des ORB zur Objektidentifikation schon immer eindeutige Objektreferenzen, diese sind allerdings nicht weltweit, sondern nur innerhalb des ORB eindeutig.
Die entfernten Operationsaufrufe des ORBs werden synchron abgewickelt, wobei eine asynchrone Kommunikation ebenfalls über Oneway-Operationsaufrufe per IDL definierbar ist. Für eine ereignisbasierte Gruppenkommunikationen reicht dies jedoch nicht aus, hierfür kann der in den CORBAservices spezifizierte EventService eingesetzt werden [OMG97].
Die Offenheit von CORBA wurde in der Version 2.0 durch ein standardisiertes Internet Inter-ORB-Protokoll (IIOP) hergestellt. Dadurch ist es möglich, auch mit den ORBs anderer Anbieter zu kommunizieren. Für die Kommunikation mit nicht OMA-konformen Systemen stehen Environment-Specific Inter-ORB Protocols (ESIOPs) zu

[2]Da CORBA anfangs der einzige standardisierte Bestandteil der OMA war, hat sich CORBA mittlerweile auch als Oberbegriff hierfür eingebürgert.

den RPCs von DCE und DCOM zur Verfügung [Red96, Sta95, APRA98].
Managementfunktionalitäten sind nur in den optionalen CORBAfacilities mit einer proprietären Architektur vorgesehen [OMG95b, KN97]. Mittlerweile arbeiten aber die OMG und ISO zusammen, und CORBA und ODP nähern sich langsam einander an, so daß CORBA in einigen Bereichen zu den ODP-Referenzplattformen zählt.

2.2 Dienstvermittlung

Frühe Verteilte Systeme sahen die Adressierung eines Prozesses direkt über den Rechnernamen, auf dem sich dann der entsprechende Prozeß befinden mußte, vor. Dieser Ansatz ist jedoch sehr unflexibel, da bereits ein Rechnerwechsel eine Änderung der verteilten Anwendung notwendig macht [Pop96, Tan95].
Dementsprechend wurden weitergehende Konzepte entwickelt, die – aufbauend auf dem Begriff des Dienstes – flexiblere Adressierungskonzepte vorsehen, über die es möglich wird, Softwarekomponenten, die unabhängig voneinander entwickelt wurden, zusammenarbeiten zu lassen. [Pop96, SPM94, Kel93] beschäftigen sich ausführlich mit dem Aspekt der Dienstvermittlung in Verteilten Systemen.

2.2.1 Dienste in Verteilten Systemen

In [SPM94] ist ein Dienst definiert als *„Funktion, die von einem Objekt an einer Schnittstelle angeboten wird“*. Zunächst handelt es sich also nur um einen abstrakten Begriff, der die von einem Objekt bereitgestellte Datenverarbeitungsfunktion bezeichnet. Im Abschnitt 2.2.3 wird das Konzept des Dienstes allerdings noch um einige Bestandteile erweitert.
Der Unterscheidung zwischen Client und Server entsprechend läßt sich zwischen Dienstnutzer und Diensterbringer unterscheiden. Ziel einer Dienstvermittlung ist es demnach, das dynamische Binden zu unterstützen, indem für einen Dienstnutzer der gewünschte Diensterbringer gefunden wird [Kov94].

2.2.2 Namensdienst

Die erste Verbesserung gegenüber einer statischen Prozeßadressierung stellen Namensdienste dar. Die Server werden hierbei statt über ihre physikalische Adresse über einen logischen bzw. symbolischen Namen angesprochen. Unter der Voraussetzung, daß dem Dienstnutzer der Name des gewünschten Dienstes bekannt ist, benötigt der Client keine weiteren Informationen über den Server: Ein Diensterbringer registriert sich bei einem systemweit verfügbaren Name-Server unter Angabe seines logischen Namens und seiner physikalischen Adresse. Ein Client kann nun mit Hilfe des logischen Namens die physikalische Adresse des Serverprozesses erfragen und mittels dieser Adresse über den dynamischen Binder den gewünschten Dienst nutzen.

2.2.3 Trading

Der Übergang vom Namensdienst zum Trader wird häufig mit dem Übergang von einem Telefonbuch zu den „Gelben Seiten“ verglichen. Während beim Namensdienst jeder Dienst einen eindeutigen Namen besitzen muß, sieht das Konzept des Tradings vor, den gewünschten Dienst über seinen Typ und seine Eigenschaften zu beschreiben. Der Trader sucht dann auf Anfrage eines Dienstnutzers (Importer) aus seiner Datenbank den Dienstanbieter (Exporter) heraus, der die gewünschten Eigenschaften besitzt bzw. – falls mehrere passende Dienstangebote existieren – denjenigen, der zusätzlich angegebene Optimalitätskriterien am besten erfüllt. Die eigentliche Dienstnutzung erfolgt im Anschluß daran ohne den Trader, indem eine Bindung zwischen Dienstnutzer und dem soeben ermittelten Dienstanbieter hergestellt wird. Der generelle Ablauf einer Dienstvermittlung über einen Trader ist in Abbildung 2.10 dargestellt [Kov94].

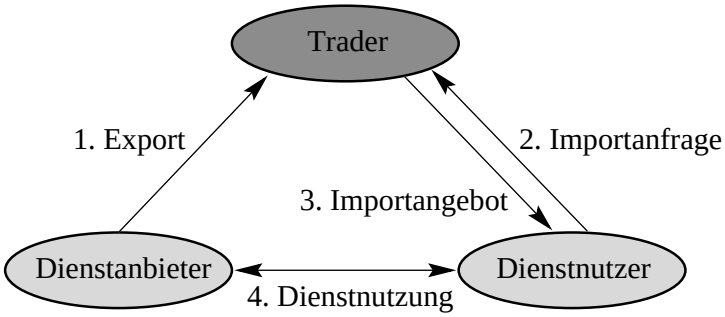

Abbildung 2.10: Ablauf der Dienstvermittlung mittels eines Traders

Während beim Namensdienst Namen für die konkreten Dienstinstanzen vergeben werden, sieht das Trading eindeutige Namen für die jeweiligen Diensttypen vor. Der Diensttyp erweitert das Konzept des Dienstes um die Angabe, welche Funktionalität ein Dienst erbringt. Aus der Angabe des Diensttyps (z.B. „Drucker“, „Compiler“) ergibt sich auch automatisch die Signatur bzw. Schnittstellendefinition eines Dienstes, da für jeden Diensttyp die zur Verfügung gestellten Operationen einheitlich festgelegt sind. Nur so ist ein offener Dienstmarkt möglich.
Unterschiedliche Dienste des gleichen Diensttyps können über Diensteigenschaften (z.B. „Papierformat“ beim „Drucker“-Dienst, „Zielplattform“ beim „Compiler“-Dienst) genauer charakterisiert werden, da die alleinige Angabe des Diensttyps oft eine zu grobe Granularität aufweist. Je nachdem, ob die Diensteigenschaft veränderlich (z.B. „Druckerwarteschlangenlänge“) oder unveränderlich (z.B. „Papierformat“) ist, wird zwischen dynamischen und statischen Attributen unterschieden.
Die Ermittlung dynamischer Dienstattribute erfordert spezielle Konzepte, um sie mit möglichst geringem Kommunikationsaufwand im Trader aktuell zu halten. Grundsätzlich kann hierbei zwischen Polling, bei dem der Trader die dynamischen Attribute der in Frage kommenden Dienste bei jeder Importanfrage ermittelt, und Caching, bei dem die exportierenden Dienste jede Attributänderung dem Trader mitteilen, unterschieden

werden. Der Ablauf beider Verfahren ist Abbildung 2.11 zu sehen. Je nach Szenario (häufige Importanfragen bei seltenen Attributänderungen im Gegensatz zu seltenen Importanfragen bei häufigen Attributänderungen) sind diese Verfahren unterschiedlich gut geeignet, wobei auch hybride Verfahren möglich sind [Kov94, Küp95].

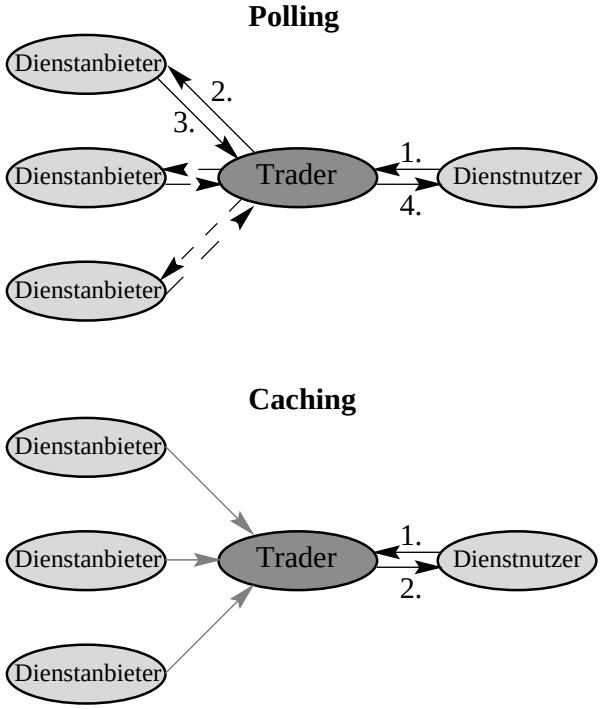

Abbildung 2.11: Anfragereihenfolge bei der Aktualisierung dynamischer Attribute

Zur weiteren Strukturierung der in der Traderdatenbank registrierten Dienstangebote sind in vielen Tradern Tradingkontexte (Dienst-Directories) vorgesehen. Ähnlich einem Verzeichnisbaum in einem Dateisystem können hierin Dienste unter administrativen Gesichtspunkten eingetragen werden.
Häufig ist auch die Zusammenarbeit von mehreren Tradern, die jeweils für einen Bereich (Domäne) zuständig sind, möglich. Das Konzept der Trader-Föderation sieht vor, daß ein Trader Import-Anfragen an andere Trader weiterreicht. Hierdurch wird die Menge der Dienste, die ein einzelner Trader vermitteln kann, vergößert, da nun zusätzlich das Angebot der anderen Trader ebenfalls zur Verfügung steht. Die genaueren Umstände der Zusammenarbeit zwischen den Tradern werden über Föderationsverträge geregelt. Auf den Aspekt der Zusammenarbeit von Tradern wird in dieser Arbeit jedoch nicht weiter eingegangen.
Im Bereich des Trading hat der ODP-Trading-Standard [ISO96] eine große Bedeutung. Er umfaßt alle oben beschriebenen Trading-Konzepte und definiert u.a. eine Trading-Schnittstelle mit Funktionen zum Im- und Export von Diensten. Die OMG hat ebenfalls einen Tradingdienst eingeführt. Dieser wurde innerhalb der CORBAservices standardisiert [OMG97] und beinhaltetet die wichtigsten Punkte des ODP-Tradings.

2.3 Lastverteilung

Ein schon recht früh eingesetztes Konzept zur schnelleren Bearbeitung von Aufgaben in Verteilten Systemen stellt die Lastverteilung dar. Während die Anfänge in der Verteilung (Scheduling) von Batch-Aufträgen in Time-Sharing-Systemen [SG94] liegen und sich über die Anwendung bei Parallelrechnern weiterentwickelt haben, ist der Einsatz in heterogenen Verteilten Systemen eine aktuelle Thematik. So beschäftigen sich zahlreiche neuere Arbeiten [Sch97a, SB97, Sch96a, Sch96b, Kov94, KRK94, Die97] mit diesem Gebiet, dennoch sind die älteren Artikel [WM85, ELZ86, MTS89] immer noch von grundlegender Bedeutung.
Prinzipiell hat die Lastverteilung zum Ziel, alle zur Verfügung stehenden Ressourcen gleichmäßig zu nutzen, indem zu bearbeitende Aufgaben auf ungenutzte bzw. nur schwach belastete Ressourcen verteilt und dort parallel abgearbeitet werden. Hierfür wird eine Instanz, der Lastverteiler (Load Balancer), benötigt, der diese Verteilung anhand einer Lastverteilungsstrategie koordiniert.
Eine optimale Lösung des Lastverteilungsproblems minimiert die mittlere Antwortzeit für die zu bearbeitenden Aufträge. Solch eine Lösung ist allerdings nur dann möglich, wenn sowohl die Laufzeit der einzelnen Anfragen als auch die Menge der Anfragen vorher bekannt sind, was in der Praxis selten der Fall ist. Darüber hinaus ist dieses Problem NP-vollständig. In realen Systemen können Verteilungsstrategien deshalb nur eine Näherung der optimalen Lösung bieten. Untersuchungen haben aber ergeben, daß bereits einfache Verfahren eine gute Näherung erreichen können [ELZ86].
Bei einem inhomogenen System, das aus unterschiedlich leistungsstarken Rechnerknoten besteht, kann sich das Ziel, eine möglichst gleichmäßige Verteilung der Gesamtlast zu erreichen und eine möglichst kurze Antwortzeit für jeden einzelnen Auftrag zu garantieren, durchaus unterscheiden. So sorgt die Zuweisung einer Aufgabe an einen leistungsschwachen Rechnerknoten zwar für eine bessere Nutzung des Gesamtsystems, die Bearbeitungszeit für diesen einzelnen Auftrag liegt dafür allerdings höher und treibt somit im Extremfall auch die über alle Aufträge gemittelte durchschnittliche Antwortzeit in die Höhe.
Damit in einem Verteilten System überhaupt eine Lastverteilung möglich ist, muß ein Dienst von mehreren Servern gleichzeitig angeboten werden. Falls hierfür mehrere Instanzen des selben Servers benutzt werden, wird von Server-Replikation gesprochen.

2.3.1 Last in Verteilten Systemen

In einem Verteilten System umfaßt der Begriff Last hauptsächlich zwei Aspekte: Neben der reinen CPU-Last ist auch die Netzlast, die beispielsweise durch die Prozeßkommunikation verursacht wird, von großer Bedeutung. Falls es um die Nutzung von Peripheriegeräten geht, kann aber darüberhinaus auch die Betrachung ihrer Belastung von Interesse sein.
Für die Beurteilung von Last wird eine Metrik benötigt, die geeignet ist, die Belastung eines Systems zu beschreiben. Ein objektiver Lastbegriff kann bei der Model-

lierung von Rechensystemen durch Warteschlangenmodelle gewonnen werden. Die wichtigsten dabei benötigten Begriffe sollen im folgenden kurz vorgestellt werden. Eine ausführlichere Einführung in die modellbasierte Leistungsbewertung von Rechensystemen findet sich in [Hav98, Bol89, SF94].
Die Bearbeitung von Aufträgen durch einen Rechnerknoten läßt sich – entsprechend Abbildung 2.12 – über ein Warteschlangenmodell analysieren. Die Bearbeitung eines Auftrags erfolgt durch einen Server, der dafür die Bedienzeit t_b benötigt. Alle weiteren Aufträge, die in dieser Zeit eintreffen, reihen sich in die Warteschlange ein und verbringen dort eine entsprechende Wartezeit. Aus der Summe dieser beiden Zeiten ergibt sich die Antwortzeit eines Auftrags. Die Zeit, die zwischen dem Eintreffen zweier Aufträge vergeht, wird als Zwischenankunftszeit t_a bezeichnet.

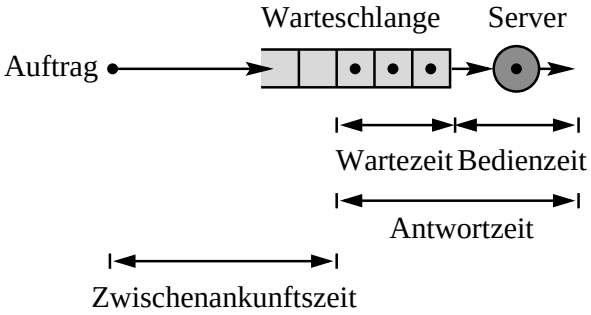

Abbildung 2.12: Ein einfaches Warteschlangenmodell

Die Last ρ ist über Verhältnis zwischen der Zeit, die der Server beschäftigt ist, und der Gesamtzeit definiert. Für das obene beschriebene Warteschlangensystem berechnet sich die Auslastung aus der über alle Aufträge gemittelten Bedien- und Zwischenankunftszeit:

$$\rho = \frac{\overline{t}_b}{\overline{t}_a} \, .$$

Soll nicht die Last eines einzelnen Servers, sondern eines Gesamtsystems betrachtet werden, bietet es sich an, statt mit mittlerer Bedien- und Zwischenankunftszeit $\overline{t}_b$ bzw. $\overline{t}_b$ mit den jeweils reziproken Bedienraten $\mu_i = \frac{1}{\overline{t}_{bi}}$ und der Ankunftsrate $\lambda = \frac{1}{\overline{t}_a}$ zu rechnen:

$$\rho = \frac{\lambda}{\sum \mu_i} \, .$$

Für ein stabiles System liegt ρ zwischen 0 und 1. Bei $\rho > 1$ ist ein System überlastet. Eine solche Überlast kann nur durch zusätzliche bzw. schnellere Server behoben werden.
Doch auch bei $\rho < 1$ können in einem zeitlich begrenzten Fenster temporäre Überlastsituationen auftreten. Dies ist dann der Fall, wenn sich Aufträge in der Warteschlange stauen und dieser Stau erst in einer Unterlastsituation abgebaut werden kann. Die

Wahrscheinlichkeit für solche Staus wächst mit steigender Last, so daß dann die mittlere Antwortzeit entsprechend ansteigt. Abbildung 2.13 stellt die theoretischen Werte für einen einzelnen Server sowie ein System mit 4 Servern dar. Die beiden Systeme wurden dazu durch eine M/M/1 und eine M/M/4-Warteschlange modelliert. Diese beiden Warteschlangenmodelle charakterisieren ein System mit 1 bzw. 4 Servern, die exponentiell verteilte Bedienzeiten haben, wobei die Zwischenankunftszeit ebenfalls exponentiell verteilt ist. Die Exponentialverteilung beschreibt einen Ankunfts- bzw. Bedienprozeß, bei dem kurze Zwischenankunfts- und Bedienzeiten wahrscheinlicher sind als lange Zwischenankunfts- und Bedienzeiten. Da die Exponentialverteilung die Eigenschaft der Gedächtnislosigkeit besitzt, ist dafür gesorgt, daß ein Ereignis unabhängig von allen vorherigen Ereignissen eintrifft. Für die Betrachtung eines Servers bedeutet dies, daß aus der Beobachtung der zurückliegenden Zwischenankunftszeiten keine Aussage über zukünftige Auftragseingänge getroffen werden kann.

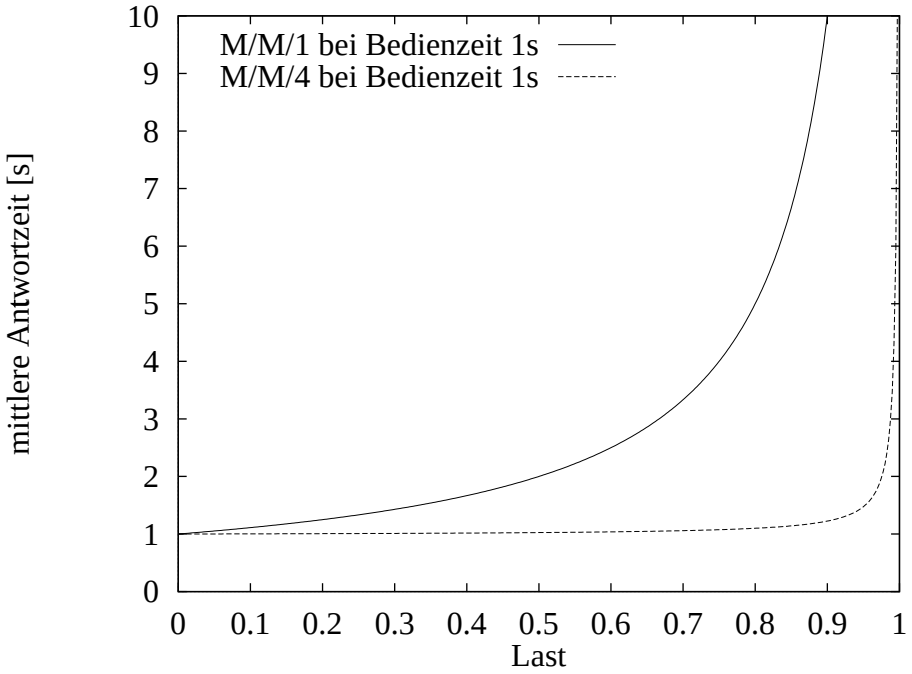

Abbildung 2.13: Mittlere Antwortzeiten in Abhängigkeit von der Last bei einem einzelnem Server und bei 4 Servern (Bedienzeit jeweils 1s)

Die abgebildeten Werte stellen die theoretischen Schranken für Lastverteilungsstrategien dar. Eine optimale Strategie kann bei k Servern nicht besser sein, als das zugehörige M/M/k-System, das bereits eine optimale Verteilungsstrategie beinhaltet. Ein M/M/1-System, bei dem lediglich ein Server genutzt wird, stellt entsprechend die Worst-Case-Schranke dar.

2.3.2 Statische Lastverteilung

Die einfachste Form der Lastverteilung stellen statische Verteilungsverfahren dar. Eingehende Aufträge werden hierbei nach einem festen Schema auf die jeweiligen Server verteilt. In der englischen Literatur wird hierfür i.d.R. der Begriff „Load Sharing“ verwendet.
Neben deterministischen Verfahren, wie z.B. zyklische Zuweisung, bei der die Last der Reihe nach auf alle Server verteilt wird, gehört auch eine zufällige (probabilistische) Verteilung auf die einzelnen Server zu den statischen Lastbalancierungsverfahren.
Statische Verfahren haben gemein, daß sie zwar einfach und effizient zu implementieren sind, sich aber nicht auf die jeweils vorliegende Situation einstellen können. Die resultierende Lastverteilung ist daher speziell in Verteilten Systemen mit inhomogener Rechenleistung ungenügend.

2.3.3 Dynamische Lastverteilung

Dynamische Lastverteilungsstrategien liefern bessere Ergebnisse als statische Verfahren, da sie sich auf die jeweiligen Situationen einstellen können. Im Englischen wird für solche adaptiven Verfahren meist der Begriff „Load Balancing“ verwendet.
Zur Anpassung an den jeweiligen Systemzustand benötigen dynamische Verfahren Informationen, die Rückschlüsse über die Last an den verschiedenen Komponenten des Systems zulassen. Auf der Grenze zwischen statischen und dynamischen Verfahren liegen die gedächtnisbehafteten Strategien, die zwar die Zahl der zurückliegenden Servernutzungen berücksichtigen, aber ansonsten über keinerlei Rückmeldung von den Servern verfügen.
Die weitergehenden dynamischen Strategien messen per Monitoring die Last an den jeweiligen Servern und beziehen diese in die weiteren Verteilungsentscheidungen ein. Bei den Quell-initiierten Verfahren kommt ein zentraler oder – um Engpässe zu vermeiden – ein auf jedem Client-Knoten replizierter Load Balancer zum Einsatz, der anhand der gesammelten Lastinformationen die eintreffenden Anfragen des Clients auf die in Frage kommenden Server verteilt.
Bei Server-initiierten Verfahren meldet sich ein Server beim Load Balancer, wenn er seine Last als niedrig einstuft, und holt von der beim Load Balancer entstehenden Warteschlange die zu bearbeitenden Anfragen ab.
Beim Entwurf einer Lastverteilungsstrategie muß berücksichtigt werden, daß die im Load Balancer gespeicherten Meßwerte, auf denen dynamische Strategien beruhen, nur die Vergangenheit wiederspiegeln können und demnach veraltet sind. Als Abhilfe wäre es denkbar, diese Daten sehr häufig an den Load Balancer zu übermitteln, um sie dort wenigstens möglichst aktuell zu halten. Dies führt jedoch zu einem erhöhten Kommunikationsaufkommen, wodurch der Load Balancer Einfluß auf die Netzlast nimmt. Der nötige Kompromiß zwischen Aktualität und Kommunikationsoverhead kann durch die Verwendung einer geeigneten Lastmetrik entschärft werden.
Neben der in Abschnitt 2.3.1 vorgestellten absoluten Metrik der Last, sind in der Praxis

weitere, relative Metriken üblich [WM85, Sch96a, Sch96b]. Gebräuchliche Metriken sind hierbei vor allem die Warteschlangenlänge, die Antwortzeit oder die Bedienrate eines Servers. Metriken, die auf der Warteschlangenlänge beruhen, haben den Vorteil, daß sie nicht nur die Vergangenheit beschreiben. Da sie die erst noch zu bearbeitenden Aufträge der Warteschlange berücksichtigen, können sie ein wenig „in die Zukunft“ blicken. Die absolute Metrik der Last hingegen basiert definitionsgemäß auf Meßwerten, die über ein zurückliegendes Zeitintervall gemittelt werden.
Für die Optimierung bezüglich der Netzlast kommt neben der reinen Netzwerkauslastung beispielsweise noch die Übertragungsrate des Netzwerksegments zwischen Client und Server – bei Weitverkehrsnetzen auch noch deren Distanz – in Betracht.
Fast alle Lastverteilungsstrategien approximieren die Lösung des Verteilungsproblems anhand einer Greedy-Heuristik, die ankommende Aufträge an den Server mit der momentan kleinsten Last weiterleitet. So verwendet die häufig eingesetzte Join-Shortest-Queue-Strategie die Warteschlangenlänge als Metrik und wählt entsprechend den Server mit der kürzesten Warteschlangenlänge aus. Eine Variante davon betrachtet dabei nicht alle Server, sondern nur einige zufällig ausgewählte Server, so daß nicht für alle Server die enstprechende Lastinformation eingeholt werden muß.
Falls die Bediendauer der einzelnen Aufträge von vorneherein bekannt ist, kann bei Server-initiierten Strategien darüberhinaus von der üblichen First-Come-First-Served-Abarbeitung der Warteschlange abgewichen werden. Stattdessen bietet sich eine Shortest-Job-First-Abarbeitungsstrategie an, durch die – unabhängig von der eigentlichen Lastverteilung – die mittlere Antwortzeit der in der Warteschlange befindlichen Anfragen verringert wird.

Kapitel 3

Optimierung der Dienstauswahl durch Lastbalancierung

In diesem Kapitel erfolgt eine Vorstellung von Konzepten für eine verbesserte Dienstauswahl durch Berücksichtigung der bei den jeweiligen Dienstanbietern vorliegenden Last. Entsprechend den Zielen der Lastverteilung sollen dadurch Leistungsengpässe bei der Dienstausführung vermieden werden, indem beim Trading nach Möglichkeit nur solche Dienstangebote vermittelt werden, die zu diesem Zeitpunkt wenig ausgelastet sind.
Nach der einführenden Behandlung existierender Arbeiten zu dieser Thematik sowie den zugrundeliegenden Gebieten des Tradings und der Lastverteilung werden weitergehende Verbesserungsmöglichkeiten für einen Trader, der bei der Dienstvermittlung die Serverlast berücksichtigt, aufgezeigt. Hieraus ergeben sich einige Anforderungen an einen verbesserten Entwurf, die im Anschluß daran vorgestellt werden. Bevor die im nächsten Kapitel beschriebene Realisierung eines verbesserten Tradingsystems vorgenommen werden kann, sind noch eine Reihe von Entwurfsentscheidungen zu treffen. Diese Entscheidungen und das durch sie festgelegte Szenario ergänzen die Anforderungsbeschreibung. Zum Schluß wird ein kurzer Überblick über die in diesem Kapitel vorgestellten Ansätze gegeben.

3.1 Arbeiten im Umfeld Trading

Zunächst sollen einige Trader-Projekte vorgestellt werden. Diese Arbeiten berücksichtigen zwar bei der Dienstvermittlung nicht explizit die Last der einzelnen Dienstanbieter, sie bilden jedoch mit ihren Konzepten zur Dienstvermittlung die Grundlage für die vorliegende Arbeit, weshalb im folgenden auf sie eingangen wird. Ein besonderes Augenmerk wird dabei auf die dynamischen Dienstattribute gerichtet, da sie eine wichtige Grundlage zur Einbeziehung der sich ständig ändernden Lastaspekte in die Dienstvermittlung darstellen.
Es existieren zahlreiche Konzepte bzw. Implementierungen zur Dienstvermittlung

mittels eines Traders [Kel93]. Als bedeutendster Standard ist hierbei der im ODP-Referenzmodell definierte ODP-Trader [ISO96] zu nennen. Die wichtigsten dort definierten Funktionalitäten wurden bereits in Abschnitt 2.2.3 beschrieben. Hierzu gehört insbesondere die Unterstützung von statischen und dynamischen Dienstattributen.
Der in den CORBAservices [OMG97] vorgesehene Trader wurde weitestgehend in Einklang mit dem ODP Trading-Modell spezifiziert. Dementsprechend sind auch hier feste und veränderliche Diensteigenschaften vorgesehen. Die Unterstützung von dynamischen Dienstattributen ist allerdings den jeweiligen CORBA-Traderimplementierungen freigestellt, so daß erst zur Laufzeit durch Nachfrage beim jeweiligen Trader ermittelt werden kann, ob diese Funktionalität angeboten wird.
Die Aktualisierung von veränderlichen Diensteigenschaften kann sowohl per Polling als auch über Caching durchgeführt werden. Die Festlegung auf eine dieser beiden Aktualisierungsstrategien obliegt den Dienstexportern, wobei auch hier dem Trader freigestellt ist, nur eine dieser Strategien zu unterstützen.
Für Attribute, die per Pollingstrategie aktualisiert werden, verwendet die OMG den Begriff „Dynamic Properties“. Zur Realisierung der Pollingfunktionalität muß ein Dienstanbieter beim Dienstexport eine Schnittstelle bekanntgeben, über die der Trader später den Wert der jeweiligen Diensteigenschaft abfragen kann. „Modifiable Properties“ bezeichnen dynamische Attribute, die per Caching aktualisiert werden. Dies geschieht, indem ein Dienstanbieter beim Trader eine entsprechende „Modify“-Operation aufruft.
Mittlerweile liegen für einige kommerzielle und freie CORBA-Systeme Traderimplementierungen vor [APRA98]. Beispielhaft sei hier der OrbixTrader von Iona [Ion98] genannt. Dieser wurde am australischen Centre for Distributed Systems Technology (DSTC) entwickelt. Er unterstützt statische und dynamische Diensteigenschaften, wobei beim Polling zur Geschwindigkeitsteigerung parallele Anfragen bei den einzelnen Dienstanbietern eingesetzt werden können.
Im folgenden wird auf den Trader, der am Lehrstuhl für Informatik IV entwickelt wurde, eingegangen. Da für diesen der Quellcode komplett vorliegt, bietet er sich für die in dieser Arbeit vorgesehene Trader-Erweiterung an.
Dieser Trader basiert auf einem frühen Entwurf der ODP-Trader-Spezifikation [ISO94]. Er wurde in einer Reihe von Diplomarbeiten erstellt bzw. weiterentwickelt und setzt auf Orbix 2.x und dem Betriebssystem Solaris [GGM93] von SUN auf. Die erste Implementierung [Zla96] bot zunächst nur die Grundfunktionen zum Im- und Export von Diensten. Zur Strukturierung des Dienstangebots können entsprechend der früheren ODP-Spezifikation Tradingkontexte benutzt werden. Mit [Sch97b] wurde die Implementierung um die Unterstützung von Trader-Föderationen erweitert und bezüglich der dabei erzielbaren Leistungssteigerung durch Caching untersucht. Eine intensivere Beschäftigung mit dynamischen Diensteigenschaften findet in [Küp95] statt. Dort wird beispielhaft eine sehr einfache Lastbalancierung anhand von dynamischen Attributen realisiert. [Thi96] behandelt die Optimierung der Dienstauswahl bei mehreren, die Dienstgüte beschreibenden Attributen. Eine eingehende Analyse der bei der Dienstvermittlung auftretenden Kommunikationslast wird in [Hei95] gegeben.

3.2 Arbeiten im Umfeld Lastbalancierung

Bevor im nächsten Abschnitt auf die Kombination von Trading und Lastbalancierung eingegangen wird, werden hiervon losgelöst zunächst einige grundsätzliche Abhandlungen des Themas Lastbalancierung betrachtet.
Zu den grundlegenden Arbeiten sind [WM85, ELZ86, MTS89] zu zählen, sie beschäftigen sich jedoch nur mit homogenen Verteilten Systemen. [WM85] vergleicht anhand von mathematischen Überlegungen und Simulationen verschiedene Strategien bzw. Scheduling-Algorithmen zur Lastverteilung und gibt dabei den Server-initiierten Verfahren den Vorzug. [ELZ86] kommt mit ähnlichen Betrachtungen zu der Aussage, daß einfache Strategien, die wenig Kommunikationsaufwand erfordern, am besten geeignet sind. Hierbei wird Schwellwert-basierten Verfahren, bei denen zufällig ausgewählten Servern solange Aufträge zugeteilt werden, bis ihre Last einen bestimmten Schwellwert überschreitet, der Vorzug gegeben. Eine Zusammenfassung dieser Ergebnisse führt zu der Gating-Technik, bei der durch zwei Schwellwerte ein Korridor definiert wird, innerhalb dessen die Last einer Ressource optimal genutzt wird. Bei Unterschreitung des unteren Schwellwerts meldet sich der Server bei der Lastverteilungskomponente, bei Überschreitung des oberen Schwellwerts werden vom Server selber keine weiteren Aufträge mehr angenommen. [MTS89] weist allerdings nach, daß dies bei Lastinformationen, deren Alter über der Ausführungszeit der bearbeiteten Aufträge liegt, nicht mehr effektiv ist.
Komplexere Strategien wurden in jüngerer Zeit ebenfalls erörtert. So sieht [KRK94] ein Neuronales Netz für einen „lernenden“ Load Balancer vor, um Ausführungszeiten anhand von vergangenen Bediendauern im voraus zu schätzen. In [Die97] wird ein auf der Fuzzy-Entscheidungstheorie basierender Load Balancer vorgestellt, der anhand von Fuzzy-Regeln diejenige Lastverteilungsstragie auswählt, die am besten für die jeweilige Situation geeignet zu sein scheint. Diese komplexen Algorithmen widersprechen jedoch der in den älteren Arbeiten aufgestellten These von der Einfachheit der Strategien, was in der neueren Arbeit [GLL96] nochmals bekräftigt wird. Auch [Del97] kommt durch Simulationen zu dem Schluß, daß Wissen über die Ausführungszeiten der Aufträge keine bzw. nur eine minimale Verbesserung gegenüber der einfachen Entscheidung anhand der aktuell an den einzelnen Rechnerknoten vorliegenden Last bringt.
Über Konzepte zur Ermittlung der Last machen die zitierten Artikel keine Angaben. Hierfür sind Arbeiten, die sich mit Leistungsmonitoring beschäftigen, heranzuziehen. So beschäftigt sich [CPRV96] beispielsweise mit der Implementierung einer Plattform für Leistungsmanagement und dem zugrundeliegenden Monitoringsystem. Diese Plattform basiert auf auf dem OSI-Managementmodell und bezieht sich dementsprechend hauptsächlich auf die Netzwerkebene. [BU96] stellt ein Konzept für das Monitoring von CORBA-Systemen im Rahmen des MAScOTTE-Projekts vor. Die Autoren geben dabei u.a. Vorschläge für Managementinformationen, die für ein Leistungsmonitoring von CORBA-Anwendungen nützlich sind. In diesen beiden Arbeiten wird jedoch nicht auf den Aspekt der Lastverteilung eingegangen.

Die Diplomarbeit [Sem97] realisiert eine dynamische Lastverteilung für ein CORBA-System. Hierzu wird zu den vorhanden Servern und Clients eine zentrale Managementkomponente, sowie pro Server-Rechnerknoten ein Monitor und pro Client-Rechnerknoten eine Lastverteilungskomponente hinzugefügt. In Abbildung 3.1 ist die grobe Architektur sowie der Kommunikationsfluß zwischen den Komponenten des dort realisierten Ansatzes dargestellt.[3]

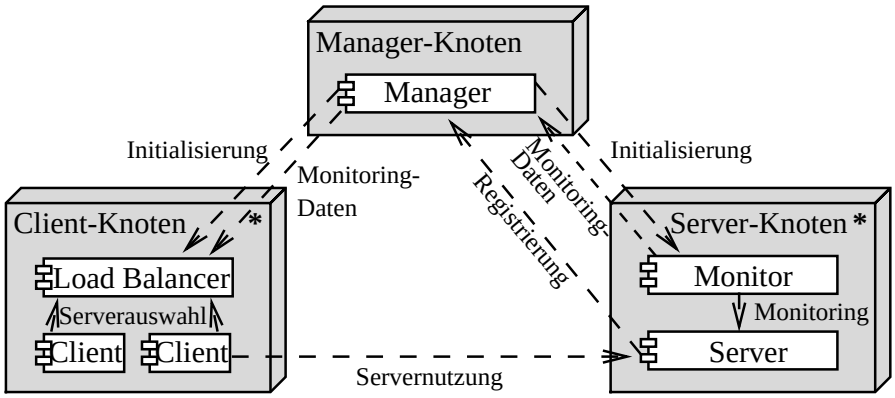

Abbildung 3.1: Verteilungsdiagramm des in [Sem97] realisierten Ansatzes

Die Ermittlung der Last erfolgt durch Meßanfragen, die ein Monitor an den lokalen Server schickt. Aus der Zeitspanne, die der Server zur Beantwortung dieser Anfrage benötigt, wird auf dessen Belastung geschlossen.
Bei dieser Arbeit werden Aspekte der Dienstauswahl nicht berücksichtigt, da die Last von lediglich einem einzelnen Dienst durch Replikation auf mehrere gleiche Serverobjekte verteilt wird. Da aber auch hierzu der Quellcode am Lehrstuhl vorliegt, wäre es denkbar, Konzepte hieraus in den bereits erwähnten Aachener Trader einfließen zu lassen.

3.3 Bestehende Ansätze der Dienstvermittlung unter dem Aspekt der Lastbalancierung

Obwohl sowohl Trading als auch Lastbalancierung immer noch ein aktuelles Forschungsthema im Bereich der Verteilten Systeme darstellen, gibt es nur wenige Arbeiten, die sich mit einer Zusammenführung dieser beiden Bereiche beschäftigen. Die Optimierung der Dienstvermittlung durch Auswahl von möglichst wenig belasteten Dienstangeboten verspricht eine Senkung der durchschnittlichen Dienstbearbeitungszeit und ist somit geeignet, die Auswirkung temporärer Leistungsengpässe zu mildern.

[3] Hierfür und für die folgenden Verteilungsdiagramme wird die Notation der Unified Modelling Language (UML)verwendet. Potentiell mehrfach vorhandene Bestandteile werden dabei mit einem „*“ gekennzeichnet.

Eine Literaturstudie lieferte im wesentlichen die im folgenden vorgestellten Projekte, die sich mit diesem Gebiet beschäftigen.
Als eine theoretische Grundlage können die Untersuchungen in [MP93, WT93] dienen. Die dort durchgeführten Simulationen bestätigen im wesentlichen die Ergebnisse der frühen Arbeiten zum Thema Lastverteilung. In beiden Arbeiten wird mit dem Konzept einer „sozialen“ Dienstauswahlstrategie gearbeitet, bei der während der Dienstauswahl globale Aspekte berücksichtigt werden. Eine „soziale“ Dienstauswahlstrategie garantiert nicht jedem einzelnen Dienstnutzer eine optimale Auswahl, vielmehr wird versucht, die Optimalitätsbedingung bezüglich des Gesamtsystems einzuhalten. Für die Lastverteilung würde dies bedeuten, daß einem Client ein langsamer Server zugewiesen wird, wenn dadurch die durchschnittliche Antwortzeit insgesamt gesenkt werden kann.
[WT93] weist auch auf die Gefahr einer zwischen den Servern oszillierenden Überlastung hin, die dadurch zustande kommen kann, daß ein besonders wenig ausgelasteter Server von einem zentralen Trader alle Aufträge zugewiesen bekommt, was bei diesem Server wiederum zu einer besonders hohen Last führt. Die dadurch entlasteten Server werden daraufhin im nächsten Schritt wieder besonders stark belastet. Diese Gefahr steigt mit dem Alter der verwendeten Lastinformationen. Zur Lösung werden dynamische Verteilungstrategien, die zusätzlich um eine Zufallskomponente angereichert sind, vorgeschlagen.
Ein besonderer Schwerpunkt wird in [WT93] auf die Verwendung von Wissen gelegt, das ein Trader aufgrund früherer Vermittlungen über die Auslastung eines Servers besitzt. Für diesen Fall wird nachgewiesen, daß sich eine lediglich dies berücksichtigende zyklische Zuweisung innerhalb der in Frage kommenden Dienstangebote und eine Lastbalancierung, die auf Leistungsmonitoring basiert, in ihrer Qualität nicht wesentlich unterscheiden. Diese Simulation beruht jedoch auf der Annahme, daß die Bearbeitungszeit der jeweils vermittelten Dienste im vorhinein bekannt bzw. immer gleich ist, so daß zu jedem Zeitpunkt bestimmt werden kann, welcher Server als nächstes wieder frei wird. Ein Ausblick, wie diese Zeiten in der Praxis ermittelt werden könnten, wird allerdings nicht gegeben.
Die gleiche Arbeit betrachtet auch das Szenario konkurrierender Trader, wie es z.B. bei Trader-Föderationen vorliegt. Es wurde der Fall untersucht, daß jeder einzelne Trader nur das Wissen über die von ihm vermittelten Dienste besitzt, wodurch sich die globale Leistung der Verteilungsstrategien entsprechend verschlechtert.
Neben diesen theoretischen Arbeiten, deren Ergebnisse auf Simulationen beruhen, gibt es einige konkrete Implementierungen von speziellen Dienstvermittlungsarchitekturen, die Lastaspekte mit einbeziehen. Im wesentlichen sind das die beiden Projekte MELODY und LYDIA, die in den folgenden Abschnitten 3.3.1 und 3.3.2 vorgestellt werden.
Außer derart spezialisierten Ansätzen können aber auch herkömmliche Dienstvermittlungssrchitekturen ansatzweise zur Lastverteilung verwendet werden. Falls ein Dienstanbieter Lastinformationen als dynamisches Attribut zur Verfügung stellt, kann ein Trader anhand dessen eine Lastverteilung vornehmen. Iona schlägt dies für sei-

nen OrbixTrader vor, indem bei der Dienstauswahl bezüglich des Lastattributs optimiert wird [Ion98]. Für die neueste Orbix-Version 3.0 wird außerdem eine auf dem Namensdienst OrbixNames basierende Lastbalancierung in Aussicht gestellt. Hierzu soll es möglich sein, zu balancierende Server durch Klassenbildung zu gruppieren. Darüber, ob eine statische oder dynamische Lastverteilung vorgenommen wird, liegen keine Informationen vor.

3.3.1 MELODY

Das an der Universität Stuttgart beheimatete Projekt MELODY (Management Environment for Large Open Distributed Systems) beschäftigt sich schwerpunktmäßig mit der Dienstvermittlung und dem Management Verteilter Systeme sowie deren Zusammenarbeit, da sich wesentliche Teile ihrer Funktionalitäten ergänzen. Eine ausführliche Beschreibung des Projekts findet sich in [KB95]. Speziellere Aspekte werden beispielsweise in [Kel93, Kov94, Kov96] behandelt.
Die Architektur sieht eine Hierarchie von Trader und Managementsystem vor, bei dem der Trader auf einem Managementsystem aufsetzt und dessen Funktionen nutzt, um z.B. dynamische Attribute oder die Last eines Dienstanbieters zu ermitteln. Die Verknüpfung zwischen Trading und Management erfolgt im MELODY-System, indem dynamische Dienstattribute auf Managementattribute abgebildet werden.
Das Managenementsystem besteht aus einem Management-Agenten (MMA) pro Knoten, bei dem sich die lokalen Komponenten anmelden können. Das Monitoring erfolgt, indem die lokalen Komponenten ihre Managementinformationen dem Management-Agenten zur Verfügung stellen. Von diesem können sie auch globale Managementinformationen abrufen. Zu diesem Zweck kommunizieren die einzelnen Management-Agenten untereinander. Zur Ermittlung dynamischer Attribute arbeitet der Trader mit den Management-Agenten zusammen, indem er anhand der Abbildung von Dienstattributen auf Managementattribute, die ein Dienstanbieter beim Export angeben muß, die aktuellen Attributwerte erfragt. Die Verteilung und Zusammenarbeit der einzelnen MELODY-Komponenten ist in Abbildung 3.2 zu sehen.

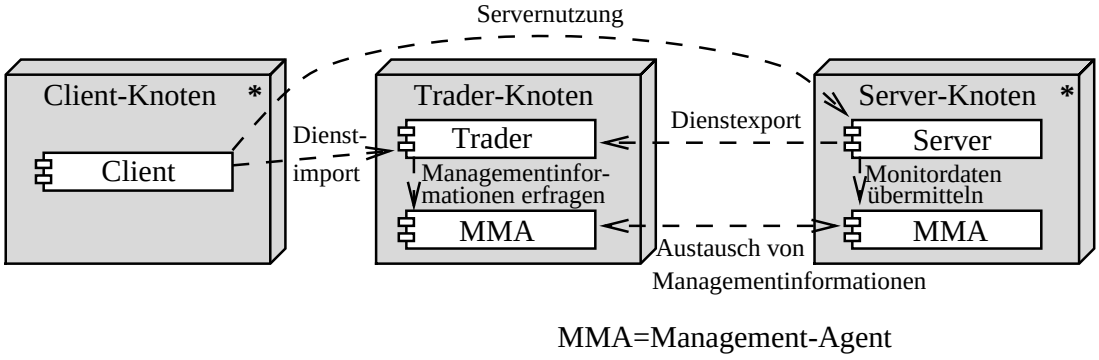

Abbildung 3.2: Verteilungsdiagramm des MELODY-Systems

Das System steht zur Zeit unter dem Betriebssystem Unix für die Verteilungsplattformen DCE und CORBA zur Verfügung. Zur Ermittlung dynamischer Attribute können verschiedene Strategien zum Einsatz kommen: Neben synchronen und asynchronen Zugriffen kann der Trader zur weiteren Geschwindigkeitssteigerung die einzelnen Zugriffe auf mehrere parallel laufende Threads verteilen. Darüberhinaus sieht das Managementsystem eine dynamische Umschaltung zwischen Caching und Polling-Strategien vor. Die Umschaltentscheidung erfolgt aufgrund der Abfrage- und Änderungshäufigkeiten der dynamischen Attribute (vgl. Abschnitt 2.2.3).
Anhand von komplexen Auswahl- und Optimierungskriterien bei der Dienstauswahl ist es möglich, eine Lastverteilung vorzunehmen. Die hierfür nötige Angabe von Gewichtungsfaktoren ist allerdings nicht standardkonform. Zur Lastbalancierung können dynamische Attribute, welche die Serverauslastung charakterisieren, verwendet werden. Diese werden vom Managementsystem entweder implizit, z.B. aus der CPU-Auslastung eines Rechnerknotens, oder explizit, z.B. über ein Warteschlangenlängen-Attribut des Servers, ermittelt.
Ein spezielles Konzept der Ressourcenreservierung erlaubt dem Trader, mit einem Dienstanbieter in Verhandlung zu treten, um dessen Ressourcen für einen gewissen Zeitraum zu reservieren, so daß einem Dienstnutzer während dieser Zeit eine bestimmte Dienstqualität garantiert werden kann. Auch dieses Konzept geht über den ODP-Tradingstandard hinaus.

3.3.2 LYDIA

Das ESPRIT-Projekt LYDIA befaßt sich mit Lastverteilung in Parallelen und Verteilten Systemen. Die wesentlichen Arbeiten, die sich dabei mit der Lastverteilung von Diensten in heterogenen Verteilten Systemen befassen, stammen von Björn Schiemann aus der Forschungs- und Entwicklungsabteilung der Siemens AG. Als Beispielanwendung dienen Transaktionssysteme für Datenbanken. In [Sch96b] wird eine detaillierte Beschreibung dieses Konzepts gegeben, [Sch96a] stellt eine Kurzfassung dieser Beschreibung dar. Neuere Ergebnisse sind in [Sch97a, SB97] zu finden.
Schiemann sieht eine nahtlose Integration der Lastverteilung für Verteilungsplattformen, die eine Interface Definition Language zur Beschreibung von Schnittstellen verwenden, vor. Dazu wird der Stub, der automatisch aus der IDL-Beschreibung erzeugt wird, um eine Monitor- bzw. Sensorkomponente erweitert. Die dort erfaßten Lastinformationen werden an einen Load Balancer weitergeleitet, der von einem Namensdienst bei der Dienstvermittlung befragt wird, um einen optimalen Dienstanbieter herauszusuchen. Hierin liegt eine Schwäche dieses Konzepts, da die Lastverteilung nur für ein Namensdienst-basiertes System konzipiert ist. Die zusätzlichen Aspekte, die sich durch den Einsatz eines Traders ergeben, wenn beispielsweise die Dienstauswahl des Traders und des Load Balancers zusammengeführt werden müssen, bleiben deshalb unberücksichtigt.
In Abbildung 3.3 ist die grobe Architektur dieses Systems dargestellt. Die vom jeweiligen Stub gemessenen Monitoringdaten werden an einen jeweils lokalen Load Balancer

Data Server geschickt und dort in einer MIB verwaltet. Dieser Data Server verbreitet die neu eingetroffenen Daten an alle anderen Load Balancer Data Server, damit jeder lokale Load Balancer Zugriff auf die globalen Lastinformationen hat. Um einen Dienst zu finden, fragt ein Client beim Namensdienst an, der dann in Absprache mit dem Load Balancer den optimalen Server zurückliefert.

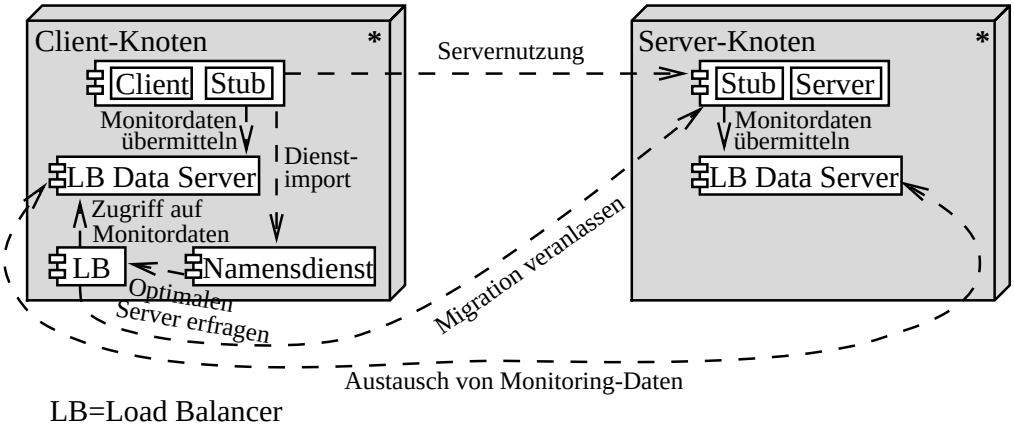

Abbildung 3.3: Verteilungsdiagramm des Systems von Schiemann

Zusätzlich unterstützt das Konzept Servermigration. Ein Load Balancer kann einen Server anweisen, die aktuelle Bindung zu unterbrechen. Dadurch wird der betreffende Client bei der nächsten Dienstnutzung gezwungen, sich direkt beim Load Balancer nach der neuen Serveradresse zu erkundigen.
Dieses Konzept ist prinzipiell auf jeder IDL-basierten Verteilungsplattform zu realisieren. Die Implementierung, die prototypisch auf einem CORBA-System erfolgte, muß jedoch bestimmte Informationen über die zugrundeliegende Plattform besitzen, da für das Monitoring der automatisch generierte Stub-Quellcode modifiziert werden muß. Dies ist als weitere Schwäche anzusehen, da diese direkte Manipulation recht unsauber ist. So muß insbesondere genau bekannt sein, wie der IDL-Compiler die Umsetzung der Schnittstellenbeschreibung auf die jeweils verwendete Programmiersprache vornimmt.

3.4 Verbesserungsmöglichkeiten

Alle vorgestellten Ansätze lösen die Problemstellung, die in dieser Arbeit behandelt werden soll, nicht vollständig. Für eine Optimierung der Dienstauswahl in einem CORBA-Trader unter dem Aspekt dynamischer Lastverteilung fehlen dort jeweils essentielle Aspekte. Die bei den einzelnen Ansätzen zu kritisierenden Punkte sind im folgenden aufgeführt.
Reine Trader-Implementierungen beschränken sich auf die Dienstvermittlung – Konzepte zur Lastverteilung sind dort nicht vorgesehen. Zwar ist es prinzipiell denkbar,

über dynamische Last-Attribute in Verbindung mit Auswahlbedingungen und Optimierungsfunktionen möglichst gering ausgelastete Dienstangebote auszuwählen; in der Praxis scheitert dies jedoch größtenteils. So ist es für einen Dienstanbieter oft problematisch, ohne einen seperaten Monitor Lastinformationen zur Verfügung zu stellen. Weit schwieriger gestaltet sich jedoch der eigentliche Auswahlprozeß innerhalb des Traders. Komplexe Lastbewertungen, die auf vielen verschiedenen Lastinformationen beruhen, sind dort nicht möglich.
Dies gilt auch für den Aachener Trader. So stellt die in [Küp95] beispielhaft vorgenommene Lastverteilung über dynamische Attribute nur eine Vorstufe zur dynamischen Lastverteilung in einem Trading-System dar. Statt kontinuierlicher Lastkennzahlen werden lediglich diskrete „Busy“ und „Idle“-Zustände, die die Servernutzung charakterisieren, verwendet. Eine eigene Lastverteilungskomponente, die eine Bewertung der unterschiedlichen Lastinformationen vornimmt und ihre Ergebnisse mit dem Trader abgleicht, ist nicht vorgesehen.
Darüberhinaus weist die vorliegende Trader-Implementierung kleinere Fehler auf, die sich u.a. bei der Benutzung von Kontexten und Diensttyp-Hierarchien zeigen. Auch die Verwendung von konkreten Objektreferenzen in Dienstangeboten muß erst noch den Eigenheiten der CORBA-Implementierung von Iona angepaßt werden. Als größtes Hindernis für eine rein auf diesem Trader basierende Lastbalancierung ist jedoch die äußerst unvollständige Unterstützung von dynamischen Diensteigenschaften anzusehen.
Die ebenfalls am Lehrstuhl für Informatik IV erstellte Diplomarbeit [Sem97] bietet zwar wiederum eine dynamische Lastverteilung über eine Lastverteilungskomponente, die von einen Managementobjekt mit den Meßdaten der Servermonitore versorgt wird, eine Dienstvermittlung ist dort allerdings nicht vorhanden. Es findet sich zwar der Hinweis auf eine einfache Integrierbarkeit in einen Trader – wie dies genau erfolgen kann, bleibt allerdings offen. An anderer Stelle wird als Motivation für den dort realisierten Lastverteilungsansatz die Replikation von Traderobjekten gegeben, wobei ebenfalls offen bleibt, wie die Zusammenarbeit der replizierten Traderobjekte aussehen kann. Darüberhinaus stellt die Ermittlung der Lastinformation eine Schwachstelle dieser Arbeit dar. Sie erfolgt durch Meßanfragen an den Server, die sich zusammen mit den regulären Aufträgen in die Server-Warteschlange einreihen. Aus der Zeit, die vergeht, bis eine abgeschickte Meßanfrage beantwortet wird, zieht der Monitor Rückschlüsse über die Warteschlangenlänge. Der dabei gewonnene Wert beschreibt jedoch nicht die aktuelle Warteschlange, sondern nur die Situation zum Zeitpunkt als die Meßanfrage abgeschickt wurde. Die Lastverteilung erfolgt demnach anhand von veralteten Informationen, die sich in der Zwischenzeit entscheidend geändert haben können. Bemerkenswert ist jedoch das Ergebnis, daß sich Server-initiierte Lastverteilungsstrategien, wie sie in [WM85] vorgeschlagen werden, in der Praxis nicht bewähren. Unabhängig davon sind derartige Strategien in einem weltweiten Dienstmarkt nur schwer zu realisieren, da hierbei an mehreren Tradern einer Föderation Warteschlangen mit Aufträgen entstehen, die alle vom Server berücksichtigt werden müßten.
Theoretische Arbeiten wie [WT93] bieten zwar ein gutes Rüstzeug für die Betrachtung

des Problembereichs der Lastverteilung in Trading-Systemen, konkrete Hilfestellung für die Implementierung der dort simulierten Strategien wird aber dadurch nicht gegeben. So ist die Verwendung von Wissen, das ein Trader über die Vermittlung von Diensten besitzt, sehr vielversprechend; die in [WT93] benötigte Aussage über die Bearbeitungszeiten von zu vermittelnden Diensten ist in der Praxis jedoch nur schwer zu realisieren. Die dort getroffene Annahme von jeweils gleicher Servergeschwindigkeit zur Schätzung von Dienstbearbeitungszeiten erscheint speziell für heterogene Systeme nicht plausibel. Denn in einem offenen Dienstmarkt sind Aussagen über die Leistungsfähigkeit der einzelnen Dienstanbieter sowie über die Komplexität eines Auftrags, bei dem der Einfluß der Eingabedaten unbekannt ist, kaum möglich.

Vielversprechender erscheinen das MELODY- und das LYDIA-Projekt. Aber auch diese lösen die Problemstellung nicht vollständig.

Das MELODY-Projekt bietet gute Ansätze aufgrund der Verknüpfung von Trading und Management samt Monitoring. In Kombination mit den angebotenen komplexen Traderanfragen und Optimierungsstrategien, die beispielsweise eine gezielte Abwägung zwischen der Qualität von Diensteigenschaften und Serverlast ermöglichen, dürften wesentliche Anforderungen der vorliegenden Problemstellung zu lösen sein. Dennoch ist das Fehlen einer speziellen Lastverteilungskomponente als Mangel anzusehen, da komplexe Lastverteilungsstrategien allein über den Trader nicht realisiert werden können. Weiterhin ist zu kritisieren, daß eine derartige Lastbalancierung nicht transparent für den Dienst-Importer erfolgt, sondern vielmehr von diesem durch Verwendung von nicht standardisierten Dienstauswahlkriterien erzwungen werden muß.

Schiemanns Arbeiten im LYDIA-Projekt bieten ein gutes Konzept zur dynamischen Lastverteilung in einem CORBA-System. Aufgrund der bereits erwähnten Schwächen genügt dieser Ansatz jedoch nicht den gestellten Anforderungen. Die Dienstvermittlung über einen Namensdienst stellt einen konzeptionellen Mangel dar. Der wünschenswerte Übergang zu einem Trader erfordert umfangreichere Änderungen, wenn das zusätzliche Potential, das sich dadurch ergibt, ausgeschöpft werden soll. Für die konkrete Realisierung ist anzumerken, daß die direkte Manipulation im Stub zwar für das übrige System extrem transparent, aber insgesamt nicht zukunftssicher ist, da zuviel Kenntnis über die zugrundeliegende CORBA-Implementierung benötigt wird. So wurde in Schiemanns Prototyp die Stub-Modifikation von Hand vorgenommen und darauf verwiesen, daß auf Interna aus dem Quelltext der Siemens-eigenen CORBA-Implementierung SORBET zurückgegriffen wird. Als sauberere Alternative führt Schiemann deshalb selber die im Orbix-System bereitgestellten Filterpunkte an, mit denen man sich über genau definierte Zugangspunkte in den Stub einklinken kann, um so ein transparentes Monitoring der über den Stub bequem zu ermittelnden Lastinformationen zu realisieren. Auch die Servermigration ist als kritisch anzusehen, da sie aufgrund der dabei zusätzlich entstehenden Last nur bei sehr lang andauernden Aufträgen von Nutzen ist, wie die Simulationen der theoretischen Arbeiten ergeben haben. Ebenso stellt sich die Frage, ob in einem offenen und heterogenen Dienstmarkt überhaupt eine Servermigration möglich ist.

3.4.1 Anforderungen an einen verbesserten Entwurf

An einen guten Ansatz zur lastabhängigen Optimierung der Dienstauswahl müssen einige konzeptionelle Anforderungen gestellt werden. Aus der vorangegangenen Analyse der Schwachstellen der existierenden Arbeiten ergeben sich die folgende Punkte, die für einen verbesserten Entwurf zu beachten sind.

- Um die Vorteile, die sich durch einen offenen Dienstmarkt ergeben, voll ausnutzen zu können, ist ein Trader zur qualifizierten Dienstvermittlung erforderlich. Unabhängig von der Optimierung durch Lastbalancierung muß es weiterhin möglich sein, die Diensteigenschaften durch Selektions- und zugehörige Optimierungskriterien bestimmen zu können.

- Zur einfachen Verwendbarkeit in bestehenden Anwendungen ist eine transparente Lastverteilung erforderlich. Der Dienstnutzer sollte sich nicht um diese neue Funktionalität kümmern müssen, sondern die gewohnten Aufrufe weiterhin verwenden können. Hierzu ist eine Lastverteilungskomponente in den Trader zu integrieren.

- Für den Einsatz in heterogenen Systemen ist die Konformität zu Standards bedeutsam. Dies bezieht sich vor allem auf die Verteilungsplattform und die Schnittstelle zum Trader.

- Neben diesen äußeren Bedingungen ist für die Interna eines verbesserten Entwurfs zu beachten, daß die Verzahnung von Trader und Load Balancer Synergieeffekte erwarten läßt. Dies ist dadurch zu begründen, daß der Load Balancer das Wissen des Traders über zurückliegende Vermittlungen benutzen kann und der gegenseitigen Zugriff auf interne Datenstrukturen eine bessere Einschätzung der Dienste hinsichtlich verschiedenster Kriterien ermöglicht.

- Im Anschluß an die Ermittlung der in Frage kommenden Dienstanbieter ist es nötig, die Ergebnisse von Trader und Load Balancer geeignet zusammenzuführen. Hierzu muß ein Regelwerk eingesetzt werden, anhand dessen bei der Zusammenführung der beiden Ergebnismengen ein Kompromiß zwischen optimalen Diensteigenschaften und geringer Last gefunden werden kann. Neben einer sinnvollen Default-Charakteristik muß es den Importern möglich sein, über die Standard-konforme Trader-Schnittstelle Einfluß auf die Parameter des Regelwerks auszuüben.

- Die Festlegung auf eine feste Lastverteilungsstrategie ist zu vermeiden, da die Erfahrung aus den vorgestellten Arbeiten lehrt, daß es sich oft erst in der Praxis herausstellt, welche Strategie für eine bestimmte Konstellation am besten geeignet ist. Es muß daher leicht möglich sein, die verwendete Strategie auszutauschen.

- Für eine effektive Lastbalancierung müssen dynamische Strategien eingesetzt werden, was die Ermittlung der Last bei den in Frage kommenden Dienstanbietern nötig macht. Hierzu ist ein entsprechendes Management- bzw. Monitoringkonzept erforderlich, das außerdem flexibel genug sein muß, den Wechsel von Lastverteilungsstragien und deren unterschiedlichen Informationsbedarf zu unterstützen.

3.4.2 Entwurfsentscheidungen

Bei der Konzeption einer Realisierung, die den zuvor aufgeführten Anforderungen entspricht, sind verschiedene Umfelder, in denen das System eingesetzt werden soll, denkbar. Einerseits soll der Entwurf flexibel genug sein, um sich diesen Szenarien anpassen zu können, andererseits müssen einige Festlegungen getroffen werden, um eine effiziente und im Rahmen dieser Diplomarbeit realisierbare Implementierung zu ermöglichen.
Zur Einschränkung des Problembereichs wurden daher einige Entscheidungen getroffen, die für den weiteren Entwurf das nun beschriebene Szenario ergeben.

Verteilungsplattform: Als Verteilungsplattform wird die am Lehrstuhl installierte CORBA-Implementierung Orbix von Iona in der Version 2.3 unter dem Betriebssystem Solaris 2.6 von SUN verwendet.

Trader: Für die Realisierung der Traderfunktionalität wird der am Lehrstuhl für Informatik IV entwickelte Trader eingesetzt. Es werden die dort implementierten Trader-Schnittstellen beibehalten.

Integration des Load Balancers: Zur besseren Verzahnung des Trading- und des Lastbalancierungsprozesses wird der vorhandene Trader im Quelltext modifiziert und um den Load Balancer angereichert, so daß auf Trader-interne Daten zugegriffen werden kann. Für lediglich als ausführbare Datei vorliegende Trader, wie z.B. den OrbixTrader, wäre aber auch ein „Wrapper“-Objekt denkbar, das den Trader und den Load Balancer getrennt ausführt und anschließend deren Ergebnisse zusammenführt. Eine Nutzung von Trader-internem Wissen wäre hierbei allerdings nicht mehr möglich.

Dauer der Bindung: Es wird eine permanente Neuwahl der Dienstanbieter jeweils vor der Nutzung eines Dienstes durch die Dienstnutzer vorausgesetzt. Der umgekehrte Fall, bei dem eine einmal vom Trader an einen Client gelieferte Dienstmenge von einem Load Balancer verwaltet wird, verspricht zwar eine schnellere Dienstauswahl, diese wäre aber nicht unbedingt optimal, falls sich in der Zwischenzeit die Diensteigenschaften der Dienstanbieter geändert haben oder neue Dienstanbieter hinzukommen. In diesem Fall wäre eine Servermigration nötig – dieser Aspekt soll in dieser Arbeit jedoch nicht betrachtet werden.

Transparenz: Die Integration der Lastbalancierung in den Trader soll für den Benutzer transparent erfolgen. Um dennoch einen Einfluß auf die Lastverteilung nehmen zu können, können Charakteristika der Lastverteilung über spezielle Dienstattribute angegeben werden. Neben der Festlegung, ob die Dienstauswahl mit oder ohne Lastberücksichtigung erfolgen soll, sind weitere Attribute zur Beschreibung der Prioritäten bezüglich der Zusammenführung der von Trader- und Load Balancer-Ergebnissen vorgesehen.

Föderatives Trading: Die zugrundeliegende Traderimplementierung unterstützt zwar die Trader-Föderation, die Einbeziehung dieses Aspekts sprengt jedoch den Rahmen dieser Diplomarbeit. So müßten geeignete Konzepte zur Ermittlung der Last bei Dienstanbietern in fremden Domänen entwickelt werden. Auch die Nutzung von Trader-Wissen durch den Load Balancer ist nicht mehr vollständig möglich, da der jeweils lokale Trader nur eine eingeschränkte Sicht auf die gesamte Föderation besitzt. Durch die Verwendung eines einzigen zentralen Traders besteht allerdings die Gefahr, daß dieser zum Flaschenhals wird und so den Gesamtdurchsatz des Systems herabsetzt. Doch auch die Verwendung von mehreren replizierten Tradern innerhalb einer Domäne würde sich als ähnlich komplex erweisen, wobei dann die Frage aufkommt, wer den Trader samt Lastbalancierer balanciert. Hierfür wäre ein seperates Lastverteilungskonzept nötig.

Management: In der Spezifikation der CORBAfacilities wird zwar auch auf den Bereich des Systemmanagements eingegangen, allerdings liegen hierfür noch keine Standards vor, weshalb außer wenigen herstellerspezifischen Ansätzen zur Zeit noch kein Managementsystem für die CORBA-Plattform vorliegt. Daher wird für diese Diplomarbeit ein eigenes Managementkonzept verwendet, das sich allerdings auf das Monitoring der für die Lastverteilung benötigten Daten beschränkt.

Anzahl Server pro Rechnerknoten: Entscheidend für das Monitoring, da bei nur einem Dienstanbieter pro Rechnerknoten jeder Monitor eindeutig mit einem Server identifiziert werden kann. Bei mehrere Servern pro Knoten muß der Monitor mehrere Server verwalten und unterscheiden, was dann auch im Load Balancer berücksichtigt werden muß. Um das Konzept flexibel zu gestalten, werden mehrere Server pro Monitor unterstützt.

Neben diesem groben Rahmen, der den zu behandelnden Problembereich einschränkt, treten noch eine Reihe weiterer Entscheidungen auf, die aber eher Details des Entwurfs betrefffen. Hierbei handelt es sich zum Teil aber um Parameter, durch die die Qualität der angestrebten Optimierung entscheidend beeinflußt werden kann, weshalb einige von ihnen bewußt offen gelassen wurden, um mehrere Alternativen testen und bewerten zu können. Hierzu zählen:

Lastbegriff: Die betrachtete Last kann vielfältigt sein, z.B. die CPU-Last, die Netzlast, die Last, die sich durch Input-/Output-Operationen ergibt, oder auch die

Last, die bei einem zu nutzenden Peripheriegerät vorliegt. In dieser Arbeit erfolgt eine Beschränkung auf die CPU-Last.

Lastmetrik: Zur Vergleichbarkeit der von den einzelnen Monitoren gelieferten Lastdaten wird eine jeweils einheitliche Metrik benötigt. Als vergleichbare Maße bieten sich die aktuelle Warteschlangenlänge, die durchschnittliche Dienstbearbeitungszeit oder Rate der Dienstnutzungswünsche pro Zeiteinheit an. Die Verwendung von absoluten Zeitpunkten sollte vermieden werden, da sonst eine komplexe Uhrensynchronisation zwischen den einzelnen Rechnerknoten erforderlich ist. Da unklar ist, welche der vorgeschlagenen Metriken am Besten geeignet ist, ist deren Bewertung Gegenstand dieser Diplomarbeit.

Lastverteilungsstrategien: Da ebenfalls unklar ist, welche Lastverteilungsstrategie einzusetzen ist, muß ein flexibler Austausch von Strategien möglich sein. Hier liegt ein enger Zusammenhang mit den Lastmetriken vor, da unterschiedliche Strategien auch unterschiedliche Informationen benötigen. Neben verschiedenen dynamischen Strategien, die auf den oben erwähnten Metriken basieren, sollen außerdem statische Verfahren, sowie Verfahren aus dem Grenzbereich von statischer und dynamischer Lastverteilung eingesetzt werden. Hierzu zählen rein probabilistische Verfahren, sowie solche, die eine zyklische Zuweisung anhand der Anzahl der zurückliegenden Vermittlungen vornehmen. Diese Strategien sollen ebenfalls getest und bewertet werden.

Ermittlung der Last: Die Last der einzelnen Dienstanbieter soll über Sensoren in den Filterpunkten, über die man sich in den Stub der Dienstschnittstellen einklinken kann, erfolgen. So ist eine transparente Meßwertaufnahme möglich, über die die wichtigsten Lastinformationen ermittelt werden können. Diese Filterpunkte werden zwar zur Zeit nur von Orbix angeboten, da sie aber in die Version 2.2 der CORBA-Spezifikation aufgenommen wurden, ist die Nicht-CORBA-Konformität dieses Entwurfs nur von kurzer Dauer.

Plazierung der Monitore: Neben der Frage, wie die Sensoren realisiert werden, ist weiterhin zu klären, wie die Monitorobjekte, die mit dem Load Balancer kommunzieren, in das System integriert werden. Solange eine Caching-Strategie zur Übermittlung der Lastinformationen an den Load Balancer eingesetzt wird, ist es ausreichend, den Monitor zusammen mit dem Sensor in den Serverprozeß zu integrieren. Diese Lösung hat den Vorteil, daß sie für das Gesamtsystem sehr transparent ist. Falls die Lastinformationen vom Load Balancer per Polling erfragt werden sollen, ist diese Plazierung jedoch problembehaftet, da eine Antwort nur dann möglich ist, wenn der Serverprozeß nicht beschäftigt ist und auf Anfragen wartet. Deshalb bietet es sich für eine Pollingstrategie an, den Monitor als seperaten Prozeß oder zumindest als eigenen Thread zu realisieren. Alternativ hierzu könnte der Load Balancer dazu übergehen, einen Serverprozeß, der nicht schnell genug auf eine Pollinganfrage antworten kann, als überlastet anzusehen

und aus der aktuellen Bewertung heraus zu lassen. Bei einer hohen Last könnte dies aber im Extremfall dazu führen, daß gar keine Server vermittelt werden. Daher wird ein seperater Monitorprozeß verwendet. Dies hat zudem gegenüber einem Thread den Vorteil, daß ein Monitor auch bei Ausfall des Servers noch Informationen, z.B. über den Serverausfall, an das Managementsystem weiterleiten kann.

Übermittlung der Monitormeßdaten an den Load Balancer: Während die Kommunikation zwischen Sensor und Monitor lokal auf dem jeweiligen Rechnerknoten erfolgt, verursacht die Übertragung der Monitordaten an den Load Balancer eine erhöhte Netzlast. Als Kommunikationsmethode bieten sich Oneway-Operationsaufrufe oder der EventService an. Da der zu realisierende Entwurf lediglich einen zentralen Load Balancer vorsieht, wird keine Gruppenkommunikation benötigt, so daß einfache Oneway-Operationsaufrufe ausreichend sind. Diese asynchronen Aufrufe sind im Gegensatz zu herkömmlichen, synchronen Operationsaufrufen schneller und erzeugen weniger Netzlast, da kein Fehlersicherungsprotokoll mit zusätzlichen Quittungen eingesetzt wird. Im Gegenzug steigt jedoch die Fehleranfälligkeit an. Da in Orbix beim Oneway-Protokoll zumindest garantiert wird, das die Nachricht fehlerfrei auf das Netzwerk gegeben wurde, ist beim Einsatz in einem rein lokalen Netz der Verlust von Nachrichten eher zu vernachlässigen.

Caching/Polling-Strategie: Da sich diese beiden Attributierungsstrategien zur Ermittlung der Lastinformationen je nachdem, wie oft die Lastinformationen benötigt werden, unterschiedlich gut eignen, müssen sowohl Caching als auch Polling implementiert werden, um deren praktische Eignung bewerten zu können. Eine Optimierung der durch das Monitoring erzeugten Netzlast kann dann durch dynamische Umschaltung zwischen beiden Strategien erfolgen.

Verwendung von Wissen über vergangene Vermittlungen: Der Load Balancer befindet sich in der Lage, das Wissen des Traders über erfolgte Vermittlungen zur Schätzung der bei den einzelnen Dienstanbietern vorliegenden Last benutzen zu können. Da das Trading-Prinzip nur den Beginn einer Dienstnutzung beinhaltet, können dadurch allerdings keine Informationen gewonnen werden, die von der Bearbeitungsgeschwindigkeit des Dienstanbieters abhängen. Von den zuvor vorgeschlagen Metriken bleibt daher nur die Anzahl der gewünschten Dienstnutzungen, die im Trader eine Schätzung der Last zuläßt, übrig. Alle weiteren Lastinformationen können nur durch Kombination mit den Monitordaten gewonnen werden.

Beschleunigung des Lastbalancierungsablaufs: Durch Parallelisierung der Dienstauswahl im Trader und im Load Balancer kann eine unnötige Verzögerung des Dienstauswahlprozesses verhindert werden. Hierzu werden mehrere Threads

benötigt, die die Ermittlung der Lastinformationen unabhängig von der Dienstauswahl des Traders ermöglichen. Dafür benötigt der Load Balancer Einblick in den laufenden Dienstauswahlprozeß, um bereits vor Ablauf dieses Prozesses entscheiden zu können, welche Dienstanbieter für die Lastverteilung in Frage kommen.

Regelwerk zur Zusammenführung der Ergebnisse: Die Ranglisten der Dienstanbieter, die vom Trader und vom Load Balancer unter verschiedenen Gesichtspunkten erstellt wurden, müssen am Ende der Importanfrage zusammengeführt werden. Hierzu ist ein Regelwerk nötig, daß aufgrund unterschiedlicher Kriterien eine Abwägung zwischen Diensteigenschaften und Last vornimmt. Eine transparente Realisierung ist im Trader möglich, wobei ein Dienstnutzer zusätzlich Präferenzen über Attributvorgaben angeben kann. So ist die Optimierung anhand einer Norm, die den Abstand der Diensteigenschaften und der Dienstlast berechnet, denkbar. Eine flexiblere Zusammenführung wäre durch Aufruf einer Callbackfunktion möglich, die der Dienstnutzer dem Trader zur Verfügung stellt, um die beiden Ranglisten zu vereinen. Für derartige Callback-Funktionen existieren jedoch keine Standards, so daß hierdurch beim Dienstimport der Traderstandard verletzt würde.

3.4.3 Einordnung der vorgestellten Ansätze

Abschließend läßt sich ein kurzer Überblick über die in diesem Kapitel behandelten Implementierungen geben. Tabelle 3.1 faßt hierzu im wesentlichen die Kritikpunkte aus Abschnitt 3.4 zusammen und ordnet den eigenen Entwurf bezüglich der anderen vorgestellten, relevanten Ansätze ein. Dies geschieht anhand von vier – für eine Optimierung der Dienstauswahl unter dem Aspekt der Lastbalancierung wesentlichen – Punkten.
Die in einen Dienstmarkt benötigte Tradingfunktionalität wird von dem reinen Loadbalancing-Ansatz in [Sem97] nicht geboten. Auch der im LYDIA-Projekt von Schiemann verfolgte Ansatz [Sch96a, Sch96b, Sch97a, SB97] basiert lediglich auf einem Namensdienst zur Dienstvermittlung. Die übrigen Ansätze weisen diesen schwerwiegenden Mangel nicht auf.
Spezielle Lastverteilungskomponenten, die komplexe Lastbewertungen vornehmen können, sind nur in [Sem97], dem LYDIA-Projekt, sowie dem in dieser Arbeit verfolgten Ansatz enthalten. Das MELODY-System [Kel93, Kov94, KB95, Kov96] bietet über sein Managementsystem zumindest Lastinformationen an. Diese können über die in diesem Trader möglichen komplexen Auswahlregeln für eine einfache Lastbalancierung verwendet werden.
Ein Regelwerk, das die unter verschiedenen Aspekten enstandenen Ergebnisse von Trader und Load Balancer zu einem Kompromiß aus niedriger Last und hoher Dienstgüte zusammenführt, bietet nur der eigene Ansatz. Da der MELODY-Trader eine Optimierung über Gewichtungsfunktionen zuläßt, wäre hiermit ansatzweise eine

enthaltene Komponenten / Ansatz	Aachener Trader	Load Balancer nach [Sem97]	MELODY	LYDIA	Eigener Ansatz
Trader	+	-	+	-	+
Load Balancer	-	+	○[4]	+	+
Regelwerk zur Ergebniszusammenführung	-	-	○[5]	-	+
allgemeines Management	-	-	+	-	-

Tabelle 3.1: Vor- und Nachteile der einzelnen Konzepte

derartige Funktionalität zu realisieren. Die angebotenen Auswahlregeln sind allerdings nicht konform zu den gängigen Traderstandards.
Für die Lastermittlung wird ein Monitorsystem benötigt. Dieses würde sich hervorragend in ein allgemeines Management von Verteilten Systemen eingliedern. Ein umfassender Managementansatz ist allerdings nur im MELODY-System enthalten. Die anderen vorgestellten Arbeiten und auch die eigene Arbeit, beschränken sich – sofern sie sich überhaupt mit Lastverteilung beschäftigen – auf ein reines Leistungsmonitoring.

[4] Ansatzweise über Traderauswahlregeln und Lastinformationen, die von Monitoren als dynamische Attribute angeboten werden.

[5] Ansatzweise über komplexe Auswahlregeln im Trader.

Kapitel 4

Realisierung eines Tradingsystems zur Lastbalancierung

Dieses Kapitel beschreibt die Realisierung eines Traders, der eine Optimierung der Dienstauswahl aufgrund der bei den einzelnen Dienstanbietern vorliegenden Last vornimmt.
Für die Realisierung wurde zunächst der durch die im vorhergehenden Kapitel gestellten Anforderungen eingegrenzte Problembereich objektorientiert modelliert. Der dabei entstandene Entwurf ist in Abschnitt 4.1 wiedergegeben.
Daran schloß sich die Implementierung des Entwurfs in der Programmiersprache C++ unter Verwendung der CORBA-Plattform Orbix an. Einzelheiten dazu sind im Abschnitt 4.2 aufgeführt.
Zur Validierung fanden Testläufe statt, anhand derer die Funktionsfähigkeit der implementierten Klassen überprüft wurde. Die abschließende Leistungsbewertung des realisierten Systems, die zur Bewertung des entwickelten Gesamtkonzepts dient, findet sich im nächsten Kapitel.
Die beschriebene Vorgehensweise stellt allerdings nur die Endsicht des Entwicklungsprozesses dar. Die einzelnen Stufen der Softwareentwicklung wurden vielmehr mehrmals durchlaufen, so daß sich das endgültige System durch einen wiederholten Prozeß von iterativen Verfeinerungen herauskristallisierte.

4.1 Modellierung des Tradings und der Lastbalancierung

Durch die objektorientierte Modellierung sollen Abläufe und Gegenstände des Problembereichs identifiziert und auf Klassen bzw. Objekte und deren Beziehungen untereinander abgebildet werden. Im Laufe des iterativen Entwicklungsprozesses entsteht dabei die Architektur des Softwaresystems.
Zur Darstellung des entwickelten Modells wird die von der OMG standardisierte Unified Modeling Language (UML) [FS98, Bur97] in der Version 1.3 verwendet. UML

bietet verschiedene Notationen für die wichtigsten Aspekte, die ein objektorientiertes Modell beinhaltet.
Die Verteilungsdiagramme zur Notation der Komponentenverteilung wurden bereits im vorhergehenden Kapitel verwendet. In diesem Kapitel werden zusätzlich noch Use-Case-Diagramme, Sequenzdiagramme und Klassenstrukturdiagramme eingesetzt. Abgesehen von Details sind diese Diagramme intuitiv verständlich, weshalb im folgenden nur kurz ihr Verwendungszweck erläutert wird.
Bei Use-Cases handelt es sich um Anwendungsfälle, die in dem zu modellierenden System auftreten. Das Identifizieren von Abläufen stellt neben der Identifizierung von Klassen einen der ersten Schritte bei der Modellerstellung dar. Die zugehörigen Diagramme beschreiben einzelne Vorgänge sowie die daran beteiligten Akteure. Zusätzlich können Beziehungen („Benutzt“, „Erweitert“) zwischen den einzelnen Vorgängen angeben werden.
Klassenstrukturdiagramme stellen eine zentrale Notation zur Beschreibung von statischen Beziehungen zwischen den identifizierten Klassen eines Modells dar. Mit diesen Diagrammtyp können Beziehungen zwischen Klassen dargestellt werden. Hierzu zählen Assoziationen („Benutzt“) zwischen Klassen, Generalisierungen („Erbt von“) sowie Klassen-Aggregationen („Enthält“).
Durch Sequenzdiagramme kann die Interaktion zwischen Objekten erfaßt werden. Dazu wird entlang einer Zeitachse der Ablauf des Nachrichtenaustauschs zwischen Objekten dargestellt. In diesem Zusammenhang ist es auch möglich, nebenläufige Prozesse (Threads) darzustellen.
Unter Berücksichtigung der Anforderungen aus Kapitel 3 ergibt sich für das zu realisierende Tradingsystem das im weiteren beschriebene Modell. Weil das gesamte Anwendungsgebiet an sich bereits einer Modellbildung entsprungen ist, gestaltet sich die Modellierung relativ einfach.

4.1.1 Anwendungsfälle

Zunächst lassen sich als Akteure der Trader und der Load Balancer, die Monitore sowie die Dienstanbieter bzw. Server-MOs und die Dienstnutzer identifizieren. Ihre Aufgaben wurden bereits in den beiden vorangehenden Kapiteln beschrieben. Von diesen Akteuren gehen im wesentlichen sieben verschiedene Anwendungsfälle aus. Die einzelnen Vorgänge werden nachfolgend beschrieben.
Obwohl die Dienstanbieter-Rolle und die Managed Object-Rolle vom gleichen Serverobjekt eingenommen werden, wird ein Server – wie bei der Managementsicht üblich – durch ein Datenverarbeitungsobjekt und ein Management-Objekt modelliert. Diese Unterscheidung ist sinnvoll, da sich diese beiden Akteure später in unterschiedlichen Rollen wiederfinden. Die Informationen des Server-MO werden in die Management Information Base eingetragen, ein Dienstanbieter findet sich hingegen als Dienstangebot in den Tabellen des Traders wieder.
Dementsprechend werden die Fälle *Dienstexport* und *Registrierung von Servern im Managementsystem* (bzw. *Dienstabmeldung (Withdraw)* und *Deregistrierung von Ser-*

vern) als unterschiedliche Anwendungsfälle angesehen. Da laut Anforderungsspezifikation die Lastverteilung transparent erfolgen soll, darf ein Dienstanbieter nichts vom Managementsystem und dessen Monitoren wissen, so daß diese beiden Anwendungsfälle voneinander zu trennen sind.

4.1.1.1 Registrierung von Servern im Managementsystem

Abbildung 4.1 stellt den Anwendungsfall der Registrierung eines Servers bzw. dessen MO beim Managementsystem dar. Dazu wird das Server-MO nach dem Serverstart bei seinem lokalen Monitor registriert. Dieser übernimmt dann für alle MOs seines Rechnerknotens eine Agentenrolle bezüglich des übrigen Systems. Dazu legt ein Monitor einen Eintrag für jedes Server-MO in seiner lokalen Management Informationen Base an.

Da ein Load Balancer die Managementinformationen aller Serverobjekte benötigt, meldet ein Monitor die von ihm überwachten Server-MOs dem Load Balancer. Dadurch erfährt letzterer, welcher Monitor für welche Server-MOs als Agent zuständig ist. Der Load Balancer merkt sich bei der Registrierung diese Zuordnung, indem er einen entsprechenden Server-Monitor-Eintrag anlegt. Die Registrierung eines Server-MOs beim Load Balancer wird durch die Anmeldung des Server-MOs bei seinem Monitor ausgelöst.

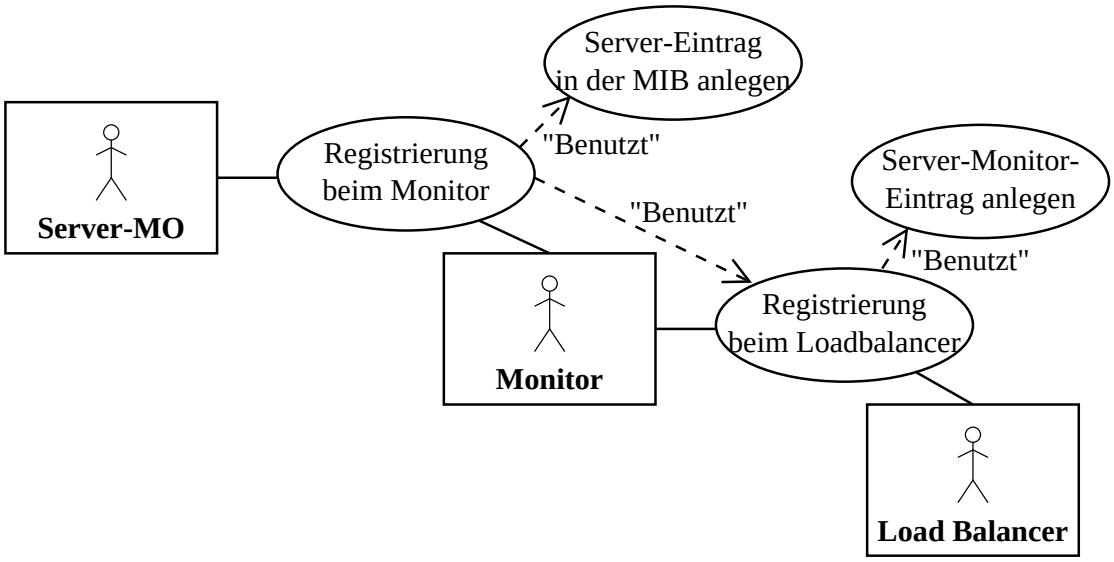

Abbildung 4.1: Use-Case-Diagramm für die Registrierung von Servern im Managementsystem

4.1.1.2 Dienstexport

Der Anwendungsfall *Dienstexport* umfaßt das Anmelden eines Dienstes bei einem Trader. Dazu wird auf den Anwendungsfall *Dienstangebot eintragen* zurückgegriffen, der das exportierte Dienstangebot in die Diensttabelle des Traders einträgt. Das zugehörige Use-Case-Diagramm findet sich in Abbildung 4.2.

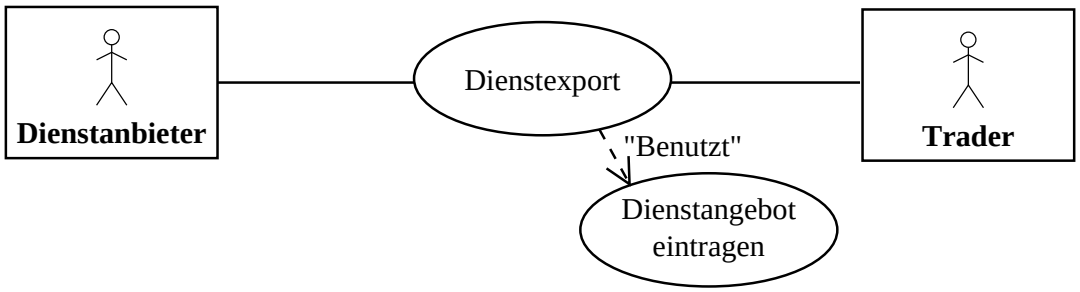

Abbildung 4.2: Use-Case-Diagramm für dem Dienstexport

4.1.1.3 Dienstimport/Lastbalancierung

In Abbildung 4.3 ist der Anwendungsfall des Imports von Diensten mit gleichzeitiger Lastbalancierung wiedergegeben. Beim Dienstimport wird zunächst ein Anwendungsfall durchlaufen, der die passenden Dienstangebote aus der Diensttabelle des Traders heraussucht. Dieser Anwendungsfall greift auf einen Typmanager zurück, um die möglichen Untertypen des gewünschten Dienstes zu ermitteln. In einem weiteren Anwendungsfall muß der Load Balancer die Last für die in Frage kommenden Server bestimmen. Wenn diese beide Anwendungsfälle durchlaufen sind, müssen die beiden Ergebnisse, die dabei entstanden sind, zusammengeführt werden.

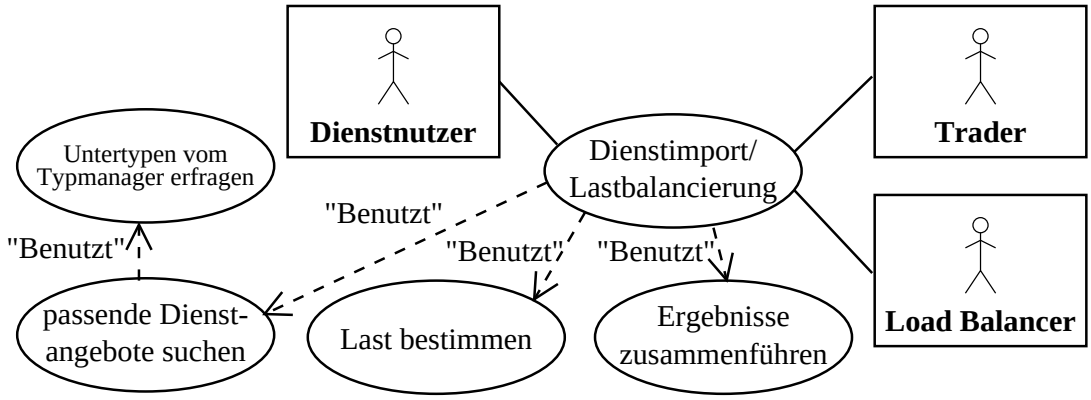

Abbildung 4.3: Use-Case-Diagramm für dem Dienstimport mit Lastbalancierung

Der Anwendungsfall *Last bestimmen*, der intern verwendet wird, greift bei Verwendung einer dynamischen Lastverteilungsstrategie auf Lastinformationen zurück, die

entweder per Caching oder Polling von den Monitoren zum Load Balancer übertragen wurden. Dieser Anwendungsfall baut daher auf dem nächsten Anwendungsfall *Lastübermittlung* auf.

4.1.1.4 Lastübermittlung

Die angesprochene Lastübermittlung ist im Anwendungsfall aus Abbildung 4.4 dargestellt. Sie verläuft in zwei Stufen. Der Übertragung der Daten von den Monitoren an den Load Balancer geht eine Notifikation, die vom Server-MO abgeschickt wird, voraus. Dabei werden die in den Sensoren des Servers angefallenen Daten an den zuständigen Monitor übertragen. Dieser berechnet aus den eingetroffenen Informationen die benötigten Lastmetriken und trägt diese in seine lokale MIB ein.
Die Lastinformationen werden in Abhängigkeit von der eingesetzten Aktualisierungsstrategie an den Load Balancer übermittelt. Bei einer Caching-Strategie wird der Anwendungsfall *Last schicken (Caching)* durchlaufen, bei einer Pollingstrategie wird vom Load Balancer der analoge Fall *Last erfragen (Polling)* angewendet. In beiden Fällen wird die übermittelte Last bis zur nächsten Aktualisierung im Load Balancer gespeichert.

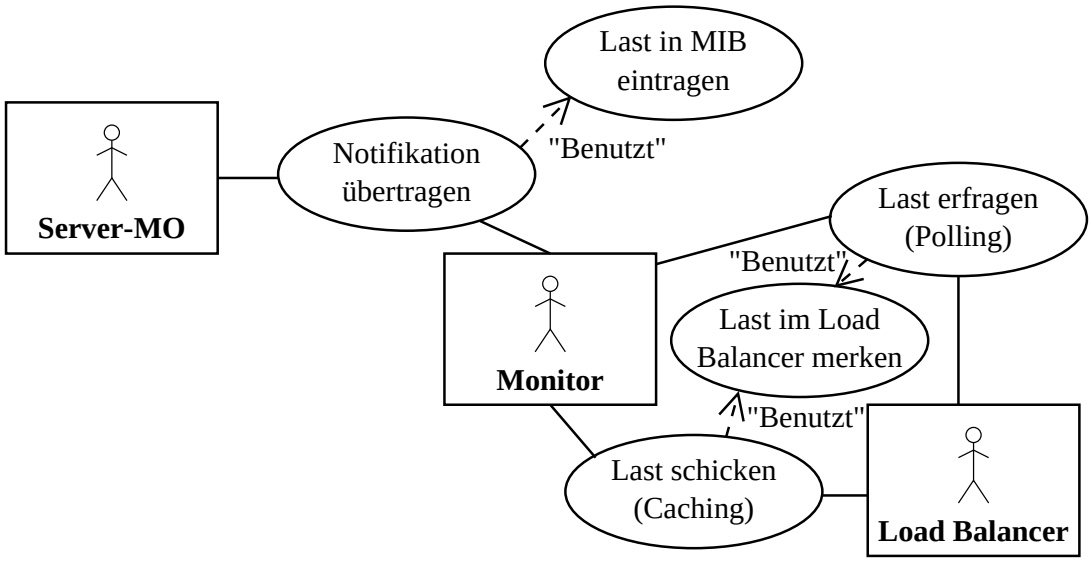

Abbildung 4.4: Use-Case-Diagramm für die Übermittlung der Last von den Monitoren zum Load Balancer

4.1.1.5 Dienstnutzung

Bei der Dienstnutzung ruft ein Dienstanbieter anhand des zuvor im Anwendungsfall *Dienstimport/Lastbalancierung* importierten Dienstangebots die gewünschte Operation eines Dienstanbieters auf.

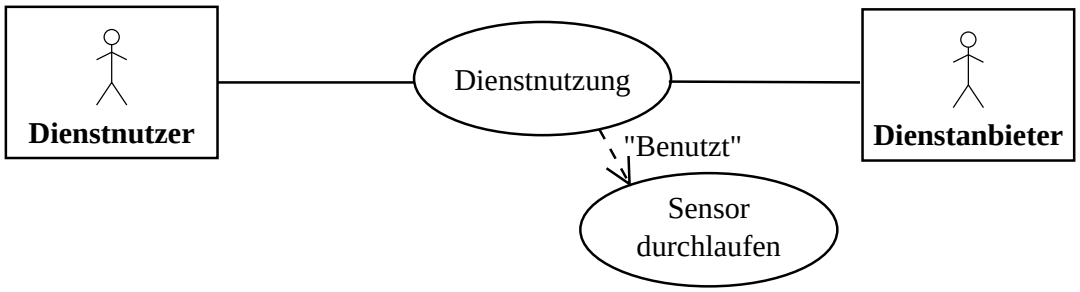

Abbildung 4.5: Use-Case-Diagramm für die Dienstnutzung

Wie der Abbildung 4.5 zu entnehmen ist, muß auf der Dienstanbieterseite neben der eigentlichen Dienstbearbeitung noch der Anwendungsfall *Sensor durchlaufen* ausgeführt werden, da sich durch die Dienstbearbeitung die Lastparameter ändern und dementsprechend neu erfaßt und als Notifikation verschickt werden müssen.

4.1.1.6 Dienstabmeldung (Withdraw)

Das in Abbildung 4.6 gezeigte Zurückziehen (Withdraw) eines Dienstangebots erfolgt vollkommen analog zum Dienstexport. Zur Abmeldung muß der beim Export vorgenommene Eintrag wieder aus der Diensttabelle des Traders entfernt werden.

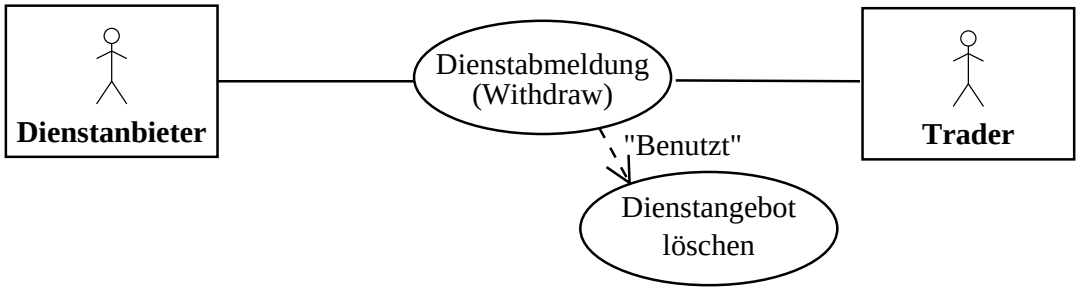

Abbildung 4.6: Use-Case-Diagramm für die Dienstabmeldung

Ein eventuelles Abmelden aus dem Managementsystem erfolgt unabhängig davon im Anwendungsfall *Deregistrierung von Servern.*

4.1.1.7 Deregistrierung von Servern

Auch die Deregistrierung von Servern läuft – wie in Abbildung 4.7 zu sehen ist – analog zur Registrierung von Servern beim Managementsystem. Die Beendigung eines Servers führt zur Deregistrierung des zugehörigen MOs. Der entsprechende Eintrag in der MIB des Monitors wird dabei gelöscht.
Im Anschluß nimmt der Monitor seine Agentenrolle gegenüber dem restlichen System war und sorgt beim Load Balancer für die Deregistrierung des abzumeldenden Server-

MO. Dies geschieht, indem der zu diesem MO gehörige Server-Monitor-Eintrag im Load Balancer gelöscht wird.

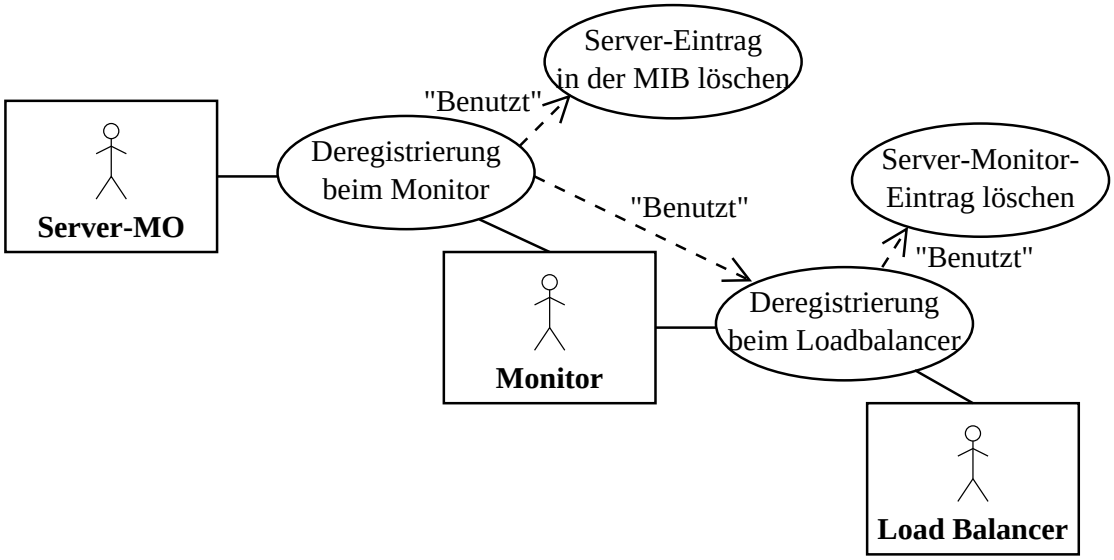

Abbildung 4.7: Use-Case-Diagramm für die Deregistrierung von Servern

4.1.2 Architektur und Verhalten des Modells

Ein Softwaremodell beinhaltet im wesentlichen zwei Aspekte: die statische Architektur und das Verhalten des Systems. Die Architektur ergibt sich aus den identifizierten Komponenten und den darin enthaltenen Klassen. Das Verhalten setzt sich aus der Funktionalität einer Klasse, sowie dem Zusammenspiel zwischen den jeweiligen Klasseninstanzen zusammen. Diese statischen und dynamischen Aspekte werden in den folgenden Abschnitten beschrieben.
Die in den Anwendungsfällen auftretenden Akteure bilden – da sie Teil des Systems sind – unmittelbar die Komponenten des Systems. Der Dienstanbieteraspekt sowie der MO-Aspekt des Servers werden allerdings zu einer einzigen Server-Komponente zusammengefaßt.
Aus ihrer Rolle innerhalb des Verteilten Systems heraus ist die Verteilung der Komponten auf die einzelnen Rechnerknoten vorgegeben. Lediglich bezüglich zweier Bestandteile des System existiert ein Entscheidungsspielraum. Die Entscheidung, die Monitore nicht als Bestandteil des Servers, sondern als seperaten Prozeß zu modellieren, wurde bereits in der Anforderungsspezifikation aus Kapitel 3 getroffen. Für den Load Balancer bietet es sich an, eine zentrale Komponente zu verwenden, da diese mit der ebenfalls zentralen Traderkomponente eng zusammenarbeiten soll.
Die konkrete Verteilung der einzelnen Komponenten ist in Abbildung 4.8 dargestellt. Sie gibt einen ersten Überblick über das erstellte Modell.

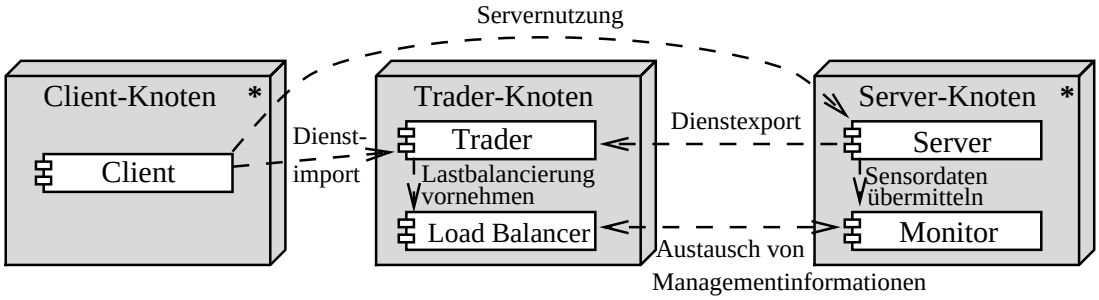

Abbildung 4.8: Verteilungsdiagramm für das Gesamtmodell

Das Modell sieht demnach für jeden Rechnerknoten, auf dem ein Server arbeitet, eine davon unabhängige Monitorkomponente zu dessen Überwachung vor. Diese Monitorkomponente steht außerdem mit der Load Balancer-Komponente, die sich auf dem gleichen Rechnerknoten wie die Traderkomponente befindet, in Verbindung. Die Traderkomponente arbeitet mit der Load Balancer-Komponente zusammen, um bei Importanfragen die Lastbalancierung vornehmen zu lassen.

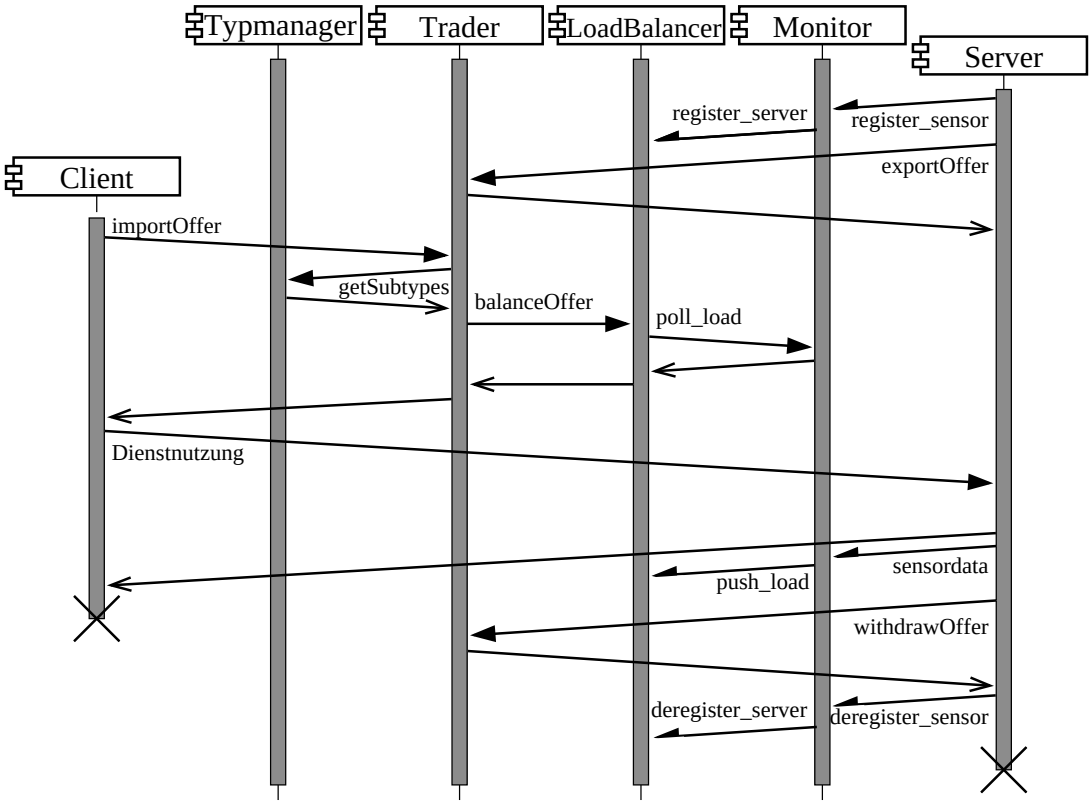

Abbildung 4.9: Sequenzdiagramm für das Gesamtsystem

Bevor die einzelnen Komponenten näher vorgestellt werden, wird deren Zusammenspiel durch das Sequenzdiagramm in Abbildung 4.9 beschrieben. Dieses Diagramm

ergibt sich unmittelbar aus den Anwendungsfällen. Ein neu gestarteter Server meldet sich zunächst innerhalb des Managementsystems an. Sein zugehöriger Monitor reicht dazu dessen Registrierung an den Load Balancer weiter. Im Anschluß daran exportiert der Server sein Dienstangebot an den Trader. Ein Client, der eine Import-Anfrage an den Trader stellt, löst dadurch eine Kommunikation des Traders mit dem Typmanager aus, da die Untertypen des gewünschten Dienstes bestimmt werden müssen. Der Trader informiert den Load Balancer über die gefundenen Dienstangebote. Dieser ermittelt dann die Last der jeweiligen Dienstanbieter und liefert die unter Lastaspekten optimierte Dienstangebotsmenge zurück. Bei der anschließenden Dienstnutzung durch den Client fallen innerhalb des Serversensors Daten an, die zum Monitor und ggf. per Caching auch an den Load Balancer übertragen werden. Vor der Terminierung zieht ein Server sein Dienstangebot beim Trader zurück und deregistriert sich innerhalb des Managementsystems.

4.1.2.1 Server

Die Serverkomponente setzt sich, wie in Abbildung 4.10 zu sehen ist, neben dem Hauptprogramm *main*, das den Server in das CORBA-System einklinkt, aus zwei grundlegenden Klassen zusammen.

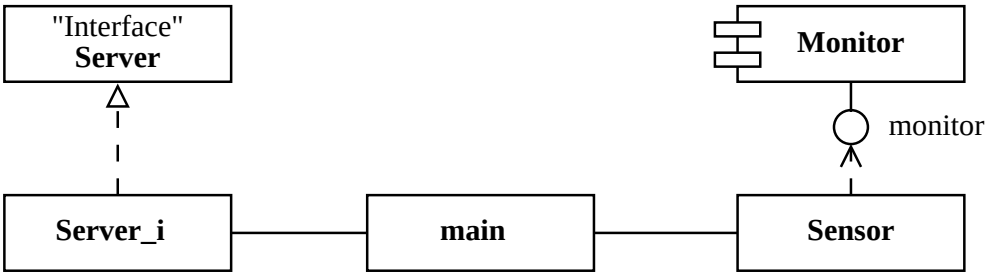

Abbildung 4.10: Klassenstrukturdiagramm der Serverkomponente

Die Klasse *Server_i* beinhaltet die eigentliche Funktionalität, die von einem Server angeboten wird. Diese Funktionalität verursacht die Last, die in dem System verteilt werden soll. Die angebotenen Operationen werden nach außen hin über die IDL-Schnittstelle *Server* angeboten. *Server_i* stellt eine Implementierung der Schnittstelle *Server* dar.
Die zweiten Klasse realisiert den Sensor, der die Lastdaten, die in diesem Server anfallen, erfaßt und über die IDL-Schnittstelle *monitor* an den lokalen Monitor weiterleitet. Neben dem Verschicken von solchen Last-Notifikationen setzt ein Sensorobjekt die Monitorkomponente auch über den Start und die Beendigung seines Serverobjekts in Kenntnis. Die konkrete Implementierung der *Sensor*-Klasse ist in Abschnitt 4.2.1 beschrieben.
Die im Sensor ermittelten Lastdaten werden über die in Abbildung 4.11 dargestellte IDL-Strukturdefinition *LoadType* repräsentiert. Dieses Format ist flexibel genug, um

im ganzen System zur Übertragung der Last eingesetzt zu werden, da es unterschiedliche Metriken vorsieht.

```
interface loadbalancing_types
{
  enum LoadmetricType {SERVICETIME_REALTIME,
    SERVICETIME_PROCESSTIME, PROCESSTIME_REALTIME_RATIO,
    QUEUELENGTH, ESTIMATED_TIME_TO_WORK, ON_IDLE, REQUEST_RATE,
    USAGE_COUNT, HOST_LOAD, UNVALID};

  struct LoadType {LoadmetricType loadmetric;
                   float          loadvalue;};
};
```

Abbildung 4.11: IDL-Spezifikation für die Lastinformationen, die im System übertragen werden

4.1.2.2 Monitor

Die Monitorkomponente muß im wesentlichen die Management Information Base für die lokalen Managed Objects verwalten und dem Load Balancer den Zugriff darauf ermöglichen. Der Anforderungsspezifikation entsprechend wird ein proprietäres Management eingesetzt, das lediglich die zu balancierenden Server und deren Last umfaßt.
Das Klassenstrukturdiagramm des Monitors ist in Abbildung 4.12 zu sehen. Die MIB, in der die Lastinformationen der lokalen Server gespeichert sind, wird über die beiden Klassen *ServerEntryList* und *ServerEntry* modelliert. Für jeden registrierten Server wird eine Instanz der Klasse *ServerEntry* angelegt und durch die Listen-Klasse verwaltet. In den einzelnen *ServerEntry*-Objekten wird die von den Sensoren übermittelte Last gespeichert. Da das Managementsystem unterschiedliche Lastmetriken unterstützen soll, werden dort alle Lastinformation, die vom Sensor in unterschiedlichen Metriken übermittelt werden, gespeichert.
Außer den Klassen, die für die Realisierung der MIB benötigt werden, wird noch die Klasse *Monitor_i* benötigt, die die *Monitor*-Schnittstelle implementiert. Die Operationen, die von den Sensoren aufgerufen werden, sind dort als „oneway“ deklariert. Da diese asynchrone Kommunikation lokal stattfindet, kann deren Geschwindigkeitsvorteil genutzt werden, ohne befürchten zu müssen, daß die Nachrichten bei der Übertragung durch Netzwerkfehler verlorengehen.
Die übrigen Operationen der Schnittstelle werden vom Load Balancer aufgerufen. *poll_load* wird verwendet, um durch Polling Lastinformationen aus der MIB des Monitors zu erfragen. Die zweite Operation schaltet von Polling auf Caching um.
Für die Kommunikation in der umgekehrten Richtung greift der Monitor wiederum auf die Schnittstelle des Load Balancers zu. Im wesentlichen handelt es sich um die Operationen zur Registrierung und Deregistrierung von Servern und zur Übertragung von Lastdaten im Caching-Betrieb.

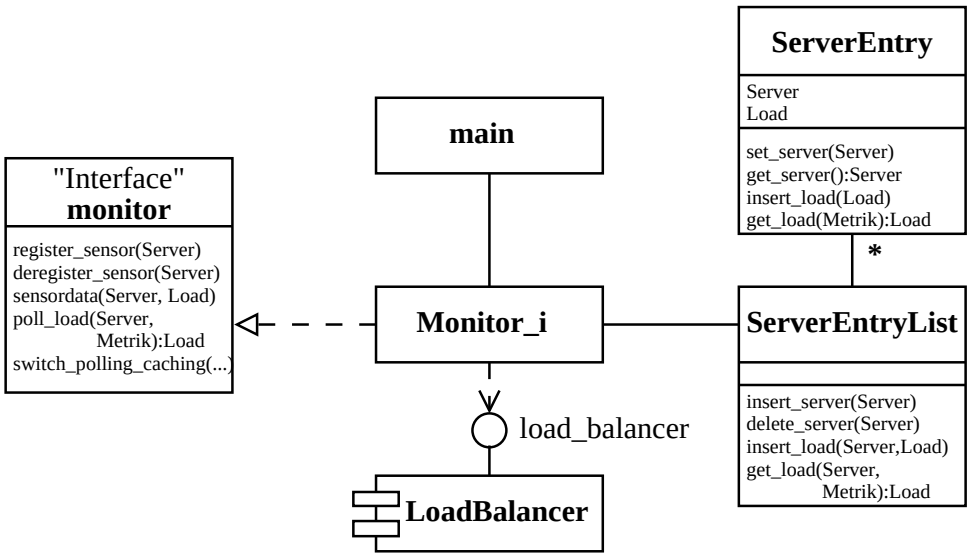

Abbildung 4.12: Klassenstrukturdiagramm der Monitorkomponente

4.1.2.3 Trader

Die Struktur der Traderkomponente konnte von der existierenden Trader-Implementierung übernommen werden. Die dort verwendete Klassenstruktur ist in Abbildung 4.13 wiedergegeben. Zusätzlich hat die Klasse *Trader_i* Zugriff auf die Klasse *Loadbalancer_i*, um dessen Lastbalancierungsoperationen aufrufen zu können, wenn passende Dienstangebote gefunden wurden.

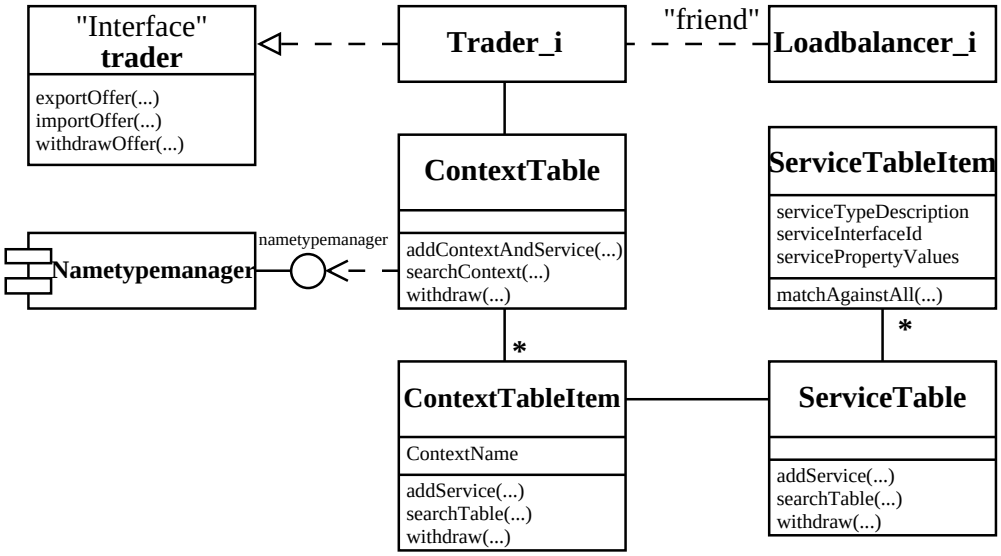

Abbildung 4.13: Klassenstrukturdiagramm der Traderkomponente

Die Klasse *Trader_i* realisiert die IDL-Schnittstelle der Tradingoperationen. Diese entsprechen der älteren ODP-Traderspezifikation [ISO94]. Aus dieser Spezifikation ergibt sich auch die Struktur der Diensttabelle, die sich aus den Klassen *ContextTable*, *ContextTableItem*, *ServiceTable* und *ServiceTableItem* zusammensetzt.
Die eigentlichen Dienstangebote sind in Instanzen von *ServiceTableItem* gespeichert. Laut der verwendeten ODP-Spezifikation werden die Dienstangebote durch Kontexte strukturiert. Jeder Kontext wird durch ein *ContextTableItem*-Objekt repräsentiert. Die in einem Kontext enthaltenen Dienstangebote sind in einem *ServiceTable*-Objekt zusammengefaßt. Bei der Suche nach einem passenden Dienstangebot (*importOffer2*) wird in der Klasse *ContextTable* die Nametypemanagerkomponente aufgerufen, um die passenden Dienstuntertypen zu bestimmen. Von den in Frage kommenden *ContextTableItem*-Objekten ausgehend wird dann nach passenden Dienstangeboten in deren Diensttabelle gesucht. Falls ein passender Eintrag gefunden wurde, trägt sich dieser in die Ergebnisliste ein.
An dieser Stelle nimmt der Trader üblicherweise die Optimierung bezüglich der Dienstattribute vor. Hierzu kann über den Parameter *matchingConstraints* ein Dienstattribut angegeben werden, das minimiert oder maximiert werden soll. Diese Stelle bietet sich auch an, um über die Friend-Klasse *loadbalancer_i* mit der Load Balancer-Komponente zusammenzuarbeiten, da in diesem Moment ein neues Dienstangebot gefunden wurde, das für die Lastbalancierung in Frage kommt. Der Load Balancer kann dann beginnen, für dieses Dienstangebot die Last zu ermitteln. Da die Dienstvermittlung unter Berücksichtigung der Last erfolgen soll, darf der Trader seine Optimierung nun nicht mehr lediglich bezüglich des Dienstattributs vornehmen, vielmehr muß er nun alle in Frage kommenden Dienstangebote zurückliefern, damit im Anschluß ein Kompromiß zwischen Last und Dienstattributierung gefunden werden kann.
Dieser Kompromiß wird über ein Regelwerk getroffen, dessen Modell in der nun folgenden Beschreibung des Load Balancers vorgestellt wird. Die genaue Implementierung wird in Abschnitt 4.2.4.4 erläutert.

4.1.2.4 Load Balancer

Die Load Balancer-Komponente benötigt für ihre Arbeit eine Tabelle, in der die für jeden Server benötigten Managementinformationen abgelegt werden. Desweiteren kommt eine Tabelle zum Einsatz, in die für jedes vom Trader gefundene Dienstangebot ein Element hinzugefügt wird. In jedem Element ist die Last des zugehörigen Servers und der Wert des zu optimierenden Dienstattributs gespeichert. Nachdem der Trader seine Diensttabellen durchlaufen hat, wird für jeden Eintrag in dieser Tabelle des Load Balancers durch ein Regelwerk eine Punktzahl berechnet, die sich aus der Last und dem Attributwert ergibt. Aufgrund dieser Punktzahl kann dann das Dienstangebot ermittelt werden, das den besten Kompromiß dieser beiden Werte darstellt.
Die genaue Klassenstruktur der Load Balancer-Komponente ist in Abbildung 4.14 zu sehen. Die Hauptklasse *Load_balancer_i* realisiert die *load_balancer*-Schnittstelle. Neben der Registrierung und Deregistrierung von Servern nimmt sie *push_load*-Aufrufe

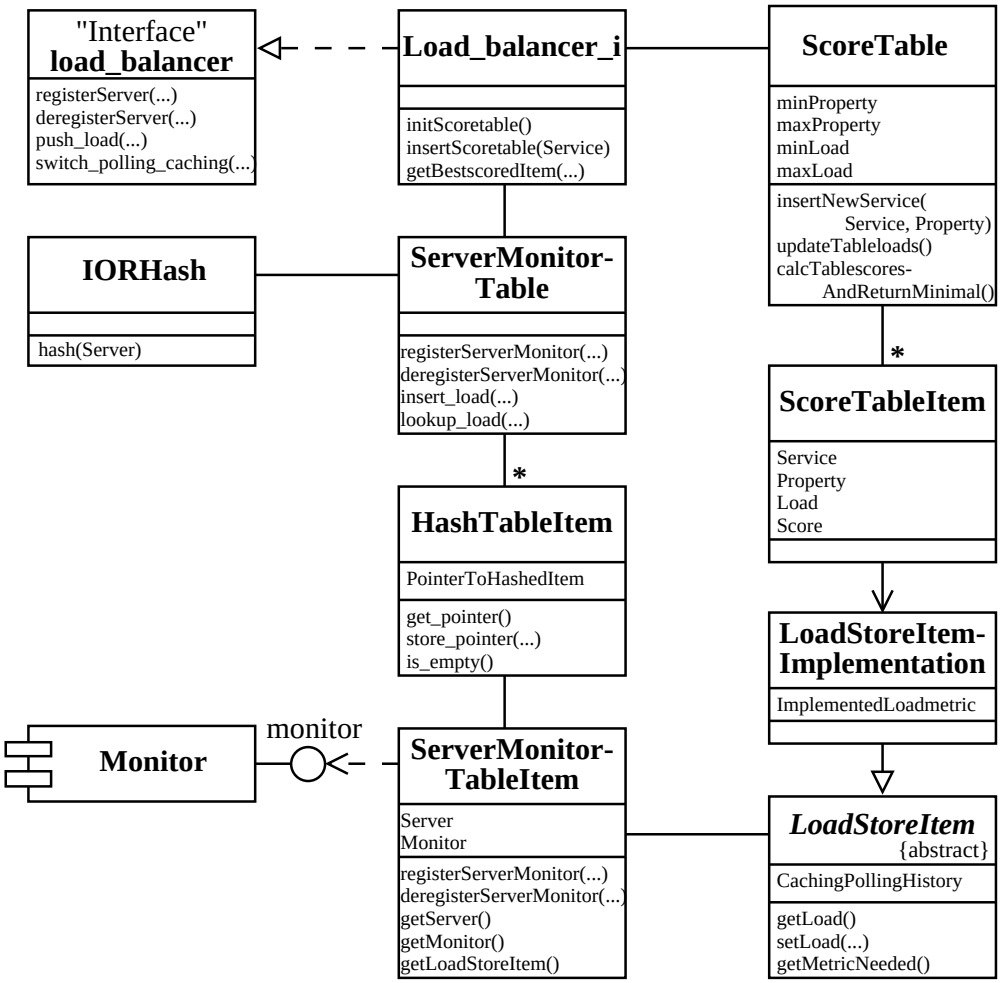

Abbildung 4.14: Klassenstrukturdiagramm der Load Balancer-Komponente

entgegen, die im Caching-Betrieb von den Monitoren an den Load Balancer geschickt werden. Darüberhinaus bietet diese Klasse Operationen an, die von der Friend-Klasse *Trader_i* aufgerufen werden, um eine Dienstangebotsmenge bezüglich der Serverlast zu balancieren.

Zur Verwaltung der Managementinformationen wird die Klasse *ServerMonitorTable* mit ihren Hilfsklassen eingesetzt. Da diese globale MIB sehr viele Einträge enthalten kann, wird der Zugriff, der anhand der Serverschnittstellen-Identifikation erfolgt, über eine Hashtabelle beschleunigt [CLR94]. Die benötigte Hashfunktion wird von der Klasse *IORHash* bereitgestellt.

Die in den Hasheinträgen gespeicherten Objekte vom Typ *ServerMonitorTableItem* enthalten zu jedem Server den Monitor, der z.B. bei Polling-Anfragen für sie zuständig ist. Die von dem jeweiligen Monitor übermittelten Lastinformationen werden in einem Objekt, das eine Implementierung der abstrakten Klasse *LoadStoreItem* bereitstellt, gespeichert. Dazu enthält ein *ServerMonitorTableItem*-Objekt einen Zeiger auf ein der-

artiges Lastspeicherungsobjekt. Für die Lastrepräsentation wurde eine abstrakte Basisklasse gewählt, um den Entwurf flexibel zu halten. Von dieser abstrakten Basisklasse sind die jeweiligen Klassen, die unterschiedliche Lastverteilungsstrategien implementieren, abgeleitet. In diesen Klassen können verschiedenen Lastmetriken zum Einsatz kommen. Das oben angesprochene Regelwerk und die abstrakte Lastklasse setzen lediglich voraus, daß das Ergebnis der jeweiligen Lastverteilungsstrategie über einen einzelnen Zahlenwert ausgedrückt werden kann.
Das Regelwerk, das einen Kompromiß zwischen Last und Dienstattributgüte finden soll, ist in der Klasse *ScoreTable* angesiedelt. Diese Klasse wird bei jeder Import-Anfrage an den Trader neu instantiiert. Die in einer solchen Instanz enthaltene Tabelle von *ScoreTableItem*-Objekten wird während der Abarbeitung der Importanfrage aufgebaut. Bei jedem passendem Dienstangebot ruft der Trader, der durch die Deklaration als „friend“ Zugriff auf den Load Balancer hat, die Methode *Load_balancer_i::insert-Scoretable* auf. Diese legt über die Methode *ScoreTable::insertNewService* ein neues *ScoreTableItem* an. Dort wird zunächst der jeweilige Dienstanbieter und dessen Wert des zu optimierenden Attributs vermerkt.
Nachdem der Trader alle passenden Dienstanbieter gefunden hat, wird die Methode *ScoreTable::updateTableloads* aufgerufen. Diese befragt die Lastverteilungsstrategie des zu dem Dienstanbieter gehörendem *LoadStoreItem* nach einem Zahlenwert, der die Last des entsprechenden Servers beschreibt, und trägt ihn in das *ScoreTableItem* ein. Dieses Aktualisierung findet erst statt, nachdem die komplette Dienstangebotsmenge feststeht, so daß alle Lastinformationen in etwa gleich alt sind. Dadurch können Änderungen der Last, die sich seit der Aufnahme eines Dienstangebots in die *ScoreTable* ergeben haben, noch berücksichtigt werden. Dies ist besonders dann von Bedeutung, wenn die Dauer einer Importanfrage sehr groß ist.
Da nun die beiden Informationen, auf denen der zu findende Kompromiß beruht, vorliegen, kann nun durch die Methode *Load_balancer_i::getBestScoredItem* bzw. *ScoreTable::calcTablescoresAndReturnMinimal* das Regelwerk aufgerufen werden. Als Ergebnis wird der Dienst zurückgegeben, der unter Verwendung der jeweils implementierten Kompromiß-Strategie die beste Punktebewertung erhalten hat. Eine Beschreibung der implementierten Strategie wird in Abschnitt 4.2.4.4 gegeben.

4.1.2.5 Client

Die Modellierung von Clients liegt außerhalb der Aufgabenstellung dieser Arbeit. An dieser Stelle soll deshalb nur kurz darauf eingegangen werden, daß es einem Clients laut Anforderungsspezifikation möglich sein soll, Einfluß auf die Parameter des Load Balancers zu nehmen.
Da der Einsatz des Load Balancers transparent innerhalb des Import-Vorgangs erfolgen soll, müssen die gewünschten Parameter über die Import-Operation der Trader-Schnittstelle angegeben werden.
Der neueste ODP-Standard [ISO96] sieht für die Import-Operation u.a. die Angabe von *scopingCriteria* vor, anhand derer Bedingungen an die Import-Operation des Tra-

ders gestellt werden können. Dies wäre auch der geeignete Ort, um Vorgaben für den Load Balancer und dessen Regelwerk zu machen.
In der Import-Operation des zugrundeliegendenen Traders ist ein solcher Parameter noch nicht vorgesehen, weshalb auf andere Parameter ausgewichen werden muß.
Es bietet sich der Parameter *serviceOfferPropOfInterest* an, über den mitgeteilt werden kann, welche Dienstangebotseigenschaften ermittelt werden sollen. Die Semantik dieses Parameters wurde dahingehend ergänzt, daß durch die Angabe bestimmter Schlüsselwörter die Verwendung des Load Balancers ein- oder ausgeschaltet werden kann bzw. von den Defaultparametern des Regelwerks abweichende Parameter angegeben werden können. Näheres hierzu findet sich in der nun folgendenden Beschreibung der Implementierung des vorgestellten Modells.

4.2 Implementierung des Modells

Die Klassen des beschriebenen Modells lassen sich unmittelbar in entsprechende C++-Klassendefinitionen übertragen. Das Modell läßt jedoch auch einige Aspekte der Implementierung offen. Insbesondere die verwendeten Lastmetriken und das Regelwerk bzw. dessen Kompromiß-Strategie wurden bewußt flexibel modelliert, so daß verschiedenste Implementierungen möglich sind.
Im folgenden wird daher ein besonderer Schwerpunkt auf die hierfür tatsächlich verwendete Implementierung bzw. die zugrundeliegenden Algorithmen gelegt. Darüberhinaus wird aber auch auf einige weitere Implementierungsdetails eingegangen, die für das Anwendungsfeld der Lastbalancierung besonders interessant sind.

4.2.1 Sensor

4.2.1.1 Erfassung der Meßdaten

Die in den Servern befindlichen Sensoren müssen verschiedene Informationen erfassen, anhand derer Aussagen über die Auslastung des Servers getroffen werden können. Zur Ermittlung dieser Daten werden Filter verwendet, die das Orbix-System zur Verfügung stellt. Zu Beginn und Ende eines entfernten Methodenaufrufs wird an insgesamt acht Stellen ein derartiger Filter aufgerufen. Abbildung 4.15 zeigt einen entfernten Methodenaufruf und die Filterpunkte, die dabei durchlaufen werden. Bevor ein Methodenaufruf einen Rechnerknoten verläßt, werden die zu übertragenden Daten in ein einheitliches Format übertragen. Wenn ein entfernter Methodenaufruf an einem Rechnerknoten eintrifft, wird die einheitlichen Darstellung wieder in die interne Repräsentation umgewandelt. Dieser Mechanismus wird als Marshalling bezeichnet. Orbix bietet vor und nach jedem Marshallingvorgang einen Filterpunkt an, in den benutzereigene Objekte eingeklinkt werden können. Hierzu müssen diese von der vordefinierten Klasse *CORBA::filter* abgeleitet werden und die Methoden der betreffenden Filterpunkte neu definieren.

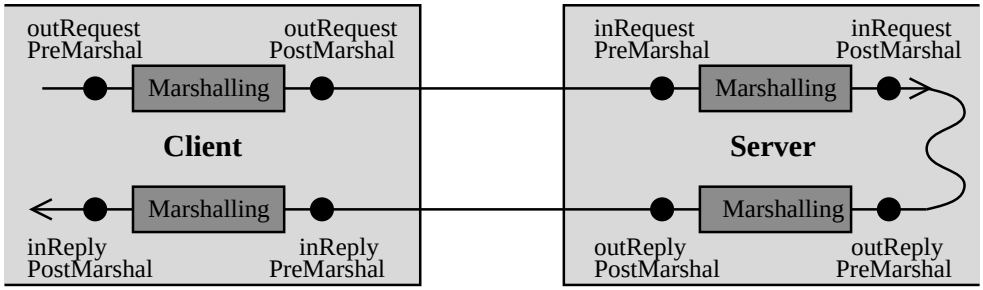

Abbildung 4.15: Die Orbix-Filterpunkte

Weil die Sensoren lediglich Daten des Servers erfassen sollen, wurden nur die Filterpunkte innerhalb des Servers verwendet. Da bereits das Marshalling eine gewisse Last verursacht, wurden die Filterpunkte *inRequestPreMarshal* und *outReplyPostMarshal* instrumentiert, um die relavanten Meßdaten zu erfassen. Tabelle 4.1 führt die Informationen, die auf diese Weise erfaßt werden, auf.

Information	Berechnung
Bedienzeit in Echtzeit	Endzeit in Echtzeit − Startzeit in Echtzeit
Bedienzeit in Prozeßzeit	Endzeit in Prozeßzeit − Startzeit in Prozeßzeit
Nutzbare CPU-Leistung	$\frac{\text{Bedienzeit in Prozeßzeit}}{\text{Bedienzeit in Echtzeit}}$

Tabelle 4.1: Meßdaten, die über die Filterpunkte erfaßt werden

Zur Zeitmessung wurden zwei verschiedene Funktionen, die Solaris 2.x zur Verfügung stellt, verwendet. Für die Messung der Echtzeit wurde *gettimeofday* benutzt. Diese Funktion liefert die aktuelle Uhrzeit in Mikrosekundenauflösung. Zur Messung der Prozeßzeit steht die Funktion *times* zur Verfügung, die eine Auflösung von 10 Millisekunden besitzt [SL94].
In Tabelle 4.1 ist die Warteschlangenlänge nicht aufgeführt, da die herkömmlichen Filterpunkte nicht geeignet sind, diese zu ermitteln. Die acht vorgestellten Filterpunkte werden nur zum Zeitpunkt der Aufrufbearbeitung durchlaufen. Aufträge, die eintreffen, während ein Server beschäftigt ist, werden in eine Orbix-eigene Warteschlange eingefügt und durchlaufen den Server-Filter erst, wenn sie der Warteschlange zur Bearbeitung entnommen werden.
Zur Lösung dieses Problems wurden Threads eingesetzt. Es werden mindestens zwei Threads benötigt. Einer der Threads bearbeitet die eigentlichen Aufträge, der zweite nimmt neu ankommende Anfragen entgegen. Für die Entgegennahme der Anfragen durch einen Thread bietet Orbix einen weiteren Filter an, der von der Klasse *CORBA::ThreadFilter* abzuleiten ist. In diesem Fall kann jedoch nicht mehr auf die Orbix-eigene Warteschlangenverwaltung zurückgegriffen werden. Stattdessen muß eine eigene Warteschlangenverwaltung implementiert werden. Hierbei kann dann auch leicht die Warteschlangenlänge ermittelt werden.

Da nebenläufige Prozesse zum Einsatz kommen, werden für den Zugriff auf gemeinsame Datenstrukturen Mechanismen zur Prozeßsynchronisation benötigt [SG94]. Solaris bietet hierfür in der Thread-Bibliothek Semaphoren und Mutexe, die einen gegenseitigen Ausschluß garantieren, an [GGM93]. Mit diesen Methoden kann das vorliegende Erzeuger-Verbraucher-Problem bequem gelöst werden. Als Erzeuger tritt der Thread, der den *ThreadFilter* ausführt und neu eintreffende Anfragen in die Wartschlange einfügt, auf. Der zweite Thread entnimmt als Verbraucher der Warteschlange Anfragen, solange diese gefüllt ist.

4.2.1.2 Kommunikation mit dem Monitor

Die in den Filterpunkten ermittelten Meßdaten werden direkt nach ihrer Erfassung an den lokalen Monitor übertragen; hierfür werden Oneway-Aufrufe verwendet. Da sich ein Server mit seinen Filterpunkten nicht im selben Betriebssystemprozeß wie der Monitor befindet, wird dieser Aufruf über den ORB abgewickelt. Eine schnellere Implementierung zur Kommunikation zwischen Sensor und Monitor wäre über Shared Memory möglich [SG94]. Aufgrund des wesentlich höheren Implementierungsaufwandes wurde diese Variante jedoch nicht verwendet. Stattdessen wurde wie – im restlichen System auch – zur Prozeßkommunikation auf CORBA zurückgegriffen.

4.2.2 Monitor

4.2.2.1 Lastmetriken in der Management Information Base

Der Monitor verwaltet in seiner MIB die von den Sensoren eingehenden Lastinformationen. Diese Informationen treffen in vier verschiedenen Metriken ein: *SERVICETIME_REALTIME, SERVICETIME_PROCESSTIME, PROCESSTIME_REALTIME_RATIO, QUEUELENGTH* (vgl. Abschnitt 4.2.1.1). Aus diesen Lastmetriken können weitere berechnet werden.
Die Monitorimplementierung verwendet für ihre proprietäre MIB neben einfachen Einträgen, in denen die von den Sensoren gelieferten Metriken gespeichert werden, auch „intelligente“ Einträge, die ihren Wert selbst bzw. aus den übrigen Einträgen berechnen können. Außer einer gleitenden bzw. exponentiellen Durchschnittsbildung für bestimmte Metriken wurde beispielhaft die im folgenden vorgestellte Metrik *ESTIMATED_TIME_TO_WORK* zur Schätzung der verbleibenden Antwortzeit implementiert.
Die exakte verbleibende Antwortzeit $T_R(t)$ eines Servers zum Zeitpunkt t ergibt sich aus den verbleibenden Bedienzeiten T_{S_i} der einzelnen im System befindlichen Aufträge J_i:

$$T_R(t) = \sum_i T_{S_i} \; .$$

Da die Bedienzeiten T_{S_i} in der Praxis allerdings nicht im voraus bekannt sind, müssen geschätzte Bedienzeiten $T^*_{S_i}$ verwendet werden. Die Schätzung der verbleibenden Antwortzeit lautet dementsprechend:

$$T_R^*(t) = \sum_i T_{S_i}^* \ .$$

In der vorliegenden Implementierung wird als geschätzte Bedienzeit $T_{S_i}^*$ für jeden Auftrag J_i der gleitende Durchschnitt $T_{\overline{S}}(t)$ verwendet, der sich aus den Bedienzeiten der zurückliegenden Aufträge ergibt. Für den Fall, daß zum Zeitpunkt t_j gerade ein Auftrag den Server verläßt und dabei noch $q(t_j)$ Aufträge in der Warteschlange stehen, kann die verbleibende Antwortzeit durch

$$T_R^*(t_j) = q(t_j) \cdot T_{\overline{S}}(t_j)$$

geschätzt werden. Für beliebige Zeitpunkte t muß berücksichtigt werden, daß sich die gesamte verbleibende Antwortzeit jeweils durch die Bearbeitung des aktuellen Auftrags verringert. Der Zeitpunkt, zu dem der momentan im Server bearbeitete Auftrag der Warteschlange entnommen wurde, sei t_j. Solange keine neuen Aufträge hinzukommen, ergibt sich als Schätzung zum Zeitpunkt t:

$$T_R^*(t) = \begin{cases} 0 & \text{falls } T_R^*(t_j) - (t - t_j) < 0 \\ T_R^*(t_j) - (t - t_j) & \text{sonst} \end{cases} \quad \text{für } t > t_j \ .$$

Falls während der Bearbeitung ein neuer Auftrag hinzukommt, erhöht sich diese Schätzung entsprechend um den Wert $T_{\overline{S}}(t)$:

$$T_R^*(t) := T_R^*(t) + T_{\overline{S}}(t) \ .$$

Das Prinzip der vorgestellten Schätzung ist in Abbildung 4.16 illustriert. Zum Zeitpunkt t_j befinden sich dort genau $q(t_j)$ Aufträge im System, so daß sich für die geschätzte verbleibende Antwortzeit der Wert $T_R^*(t_j) = q(t_j) \cdot T_{\overline{S}}(t_j)$ ergibt. Im Laufe der Zeit verringert sich dieser Wert kontinuierlich bis zum Zeitpunkt t', an dem ein neuer Auftrag die Warteschlange betritt. Während dieses Zeitraums hat sich die Schätzung um den Betrag $(t' - t_j)$ verringert. Durch den neuen Auftrag erhöht sich die Schätzung um die geschätzte Bedienzeit $T_{\overline{S}}(t')$.

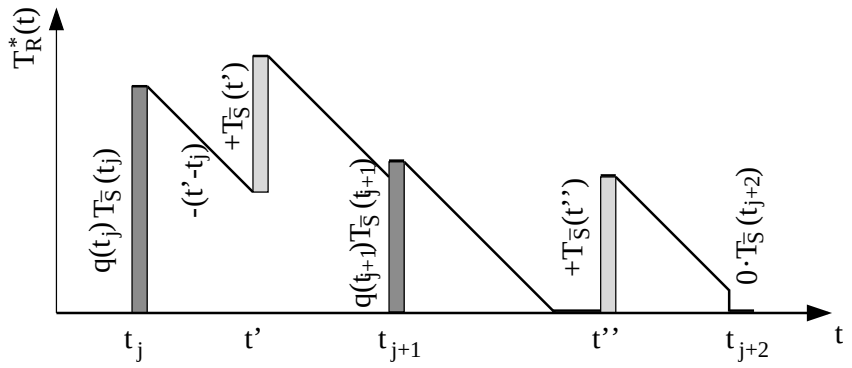

Abbildung 4.16: Schätzung der verbleibenden Antwortzeit

Zum Zeitpunkt t_{j+1} verläßt ein Auftrag den Server, so daß die geschätzte verbleibende Antwortzeit neu berechnet wird. Da die Bearbeitung des gerade abgefertigten Auftrags länger als die zuvor verwendete durchschnittliche Bedienzeit gedauert hat, erhöht sich dieser Durchschnitt. Die aus $T_{\overline{S}}(t_{j+1})$ neu berechnete Schätzung fällt daher höher aus als ursprünglich angenommen.

Im weiteren Verlauf verringert sich der Schätzwert $T_R^*(t)$ mit der Zeit bis zur unteren Grenze 0s. Zur Zeit t'' betritt nochmals ein Auftrag das System. Seine Bearbeitung wird zum Zeitpunkt t_{j+2} abgeschlossen. Da sich nun keine Äufträge mehr im System befinden, berechnet sich die geschätzte verbleibende Antwortzeit zu 0s. Der gerade bearbeitete Auftrag wurde schneller als erwartet bedient, so daß sich bei t_{j+2} ein Sprung ergibt.

Die zugehörige Implementierung ist in den Abbildungen 4.17 und 4.18 zu sehen.

```
...
delta=newqueuelength-queuelength.oldvalue;
if (delta<=0)
  // Warteschlange wurde kuerzer, d.h. Request beendet
  // Absolute Berechnung, um Fortpflanzungsfehler klein zu halten
  // und geaenderten Mittelwert der Bedienzeit ueberall einzubeziehen.
  insert_estimated_time_to_work(
                      newqueuelength*get_avg_servicetime_realtime());
else
  // Warteschlange wurde laenger, d.h. Request dazugekommen
  // Relativer Zuwachs
  insert_estimated_time_to_work(get_estimated_time_to_work()
                       +delta*get_avg_servicetime_realtime());
...
```

Abbildung 4.17: Eintragen der Schätzung der verbleibenden Antwortzeit in die MIB

```
float ServerEntry::get_estimated_time_to_work()
{
  float value=estimated_time_to_work.value;
  long end_sec_real, end_usec_real;
  myClock.get_realtime(end_sec_real, end_usec_real);
  value-=(end_sec_real-estimated_time_to_work.timestamp_sec)
    +(end_usec_real-estimated_time_to_work.timestamp_usec)/1000000.0;
  if (value<0)
    value=0;
  return value;
};
```

Abbildung 4.18: Auslesen der Schätzung der verbleibenden Antwortzeit aus der MIB

4.2.2.2 Kommunikation mit dem Load Balancer

Während die in Abschnitt 4.2.1.2 beschriebene Kommunikation zwischen Sensor und Monitor von lokaler Natur ist, erfolgt die Übertragung von Daten aus Monitor-MIB an den Load Balancer über das Netzwerk.
Die verschickten Nachrichtenpakete sind relativ klein, weil sie vergleichsweise wenige Daten enthalten. Da diese Lastdaten aber sehr häufig übertragen werden, lohnt es sich, ihr Aufkommen näher zu betrachten.
Die Informationen, auf denen die Lastbalancierung beruht, sollten möglichst aktuell sein. [MTS89, WT93] empfehlen – wie in Kapitel 3 erwähnt –, keine Lastinformationen zu verwenden, deren Alter oberhalb der Größenordnung der Bedienzeit liegt, da es sonst zu einer sich zyklisch aufschaukelnden Überlastung von einzelnen Servern kommen kann. Um dies zu verhindern, verschicken die Monitore im Caching-Betrieb zum Ende jeder Dienstnutzung die vom Load Balancer gewünschten Lastmetriken. Einige Metriken, wie z.B. die Warteschlangenlänge oder die zuvor vorgestellte *ESTIMATED_TIME_TO_WORK*-Metrik, werden außerdem auch zu Beginn einer Dienstbearbeitung aktualisiert.
Im Caching-Betrieb fallen daher bei jeder Dienstnutzung – je nach eingesetzter Metrik – ein oder zwei Nachrichten an, die an den Load Balancer übertragen werden. Beim Polling-Modus werden für jede Dienstvermittlung die benötigten Lastinformationen der in Frage kommenden Server von den jeweiligen Monitoren erfragt.
In einer geschlossenen Trading-Domäne, in der ein Server genau dann genutzt wird, wenn er vom Trader vermittelt wurde, können einige Überlegungen bezüglich des Nachrichtenaufkommens, das durch eine Polling- bzw. Caching-Strategie verursacht wird, angestellt werden. In diesem Fall ruft jede Dienstvermittlung, der sich entsprechend eine Dienstnutzung anschließt, selber eine Änderung der Last – und damit einen Aktualisierungsbedarf – hervor.
Für eine derart geschlossene Trading-Domäne ergibt sich bezüglich der hervorgerufenen Netzlast ein Vorteil für die Caching-Strategie. In einer Domäne mit n in Frage kommenden Dienstanbietern, verursacht eine Importanfrage an den Trader n Polling-Anfragen. Bei einer Caching-Strategie würde eine Importanfrage mit anschließender Dienstnutzung – in Abhängigkeit von der verwendeten Lastmetrik – ein oder zwei Caching-Updates nach sich ziehen. Spätestens bei mehr als zwei Dienstanbietern, die das gewünschte Dienstangebot exportieren, ist demnach eine Caching-Aktualisierungsstrategie für die Lastübermittlung zu bevorzugen.
Für den Fall, daß ein Server auch unabhängig von einer Tradervermittlung genutzt wird oder sich unabhängig von der Servernutzung ändernde Lastmetriken (z.B. die Hintergrundlast eines Rechnernknotens) verwendet werden, verlieren diese Überlegungen allerdings ihre Gültigkeit. Allgemeine Aussagen können hierfür nicht mehr getroffen werden.
Um dennoch selbst in diesem Fall eine optimale Aktualisierungsstrategie, die am wenigsten Nachrichtenaufkommen zur Folge hat, zu verwenden, wurde eine dynamische Umschaltung zwischen Caching- und Pollingbetrieb implementiert. In [Küp95] ist ein

Automatismus beschrieben, der dies leistet. Ein derartiger Automatismus vergleicht die Zugriffsrate und die Änderungsrate eines Attributs. Falls sich der Wert des Attributs öfter ändert als er abgefragt wird, empfiehlt sich zur Minimierung der Nachrichtenanzahl der Einsatz einer Pollingstrategie. Der umgekehrte Fall liegt vor, wenn ein Attributwert öfter abgefragt als geändert wird. In dieser Situation ist eine Aktualisierung mittels Caching optimal. Der Automatismus ist in Abbildung 4.19 durch ein Sequenzdiagramm dargestellt.

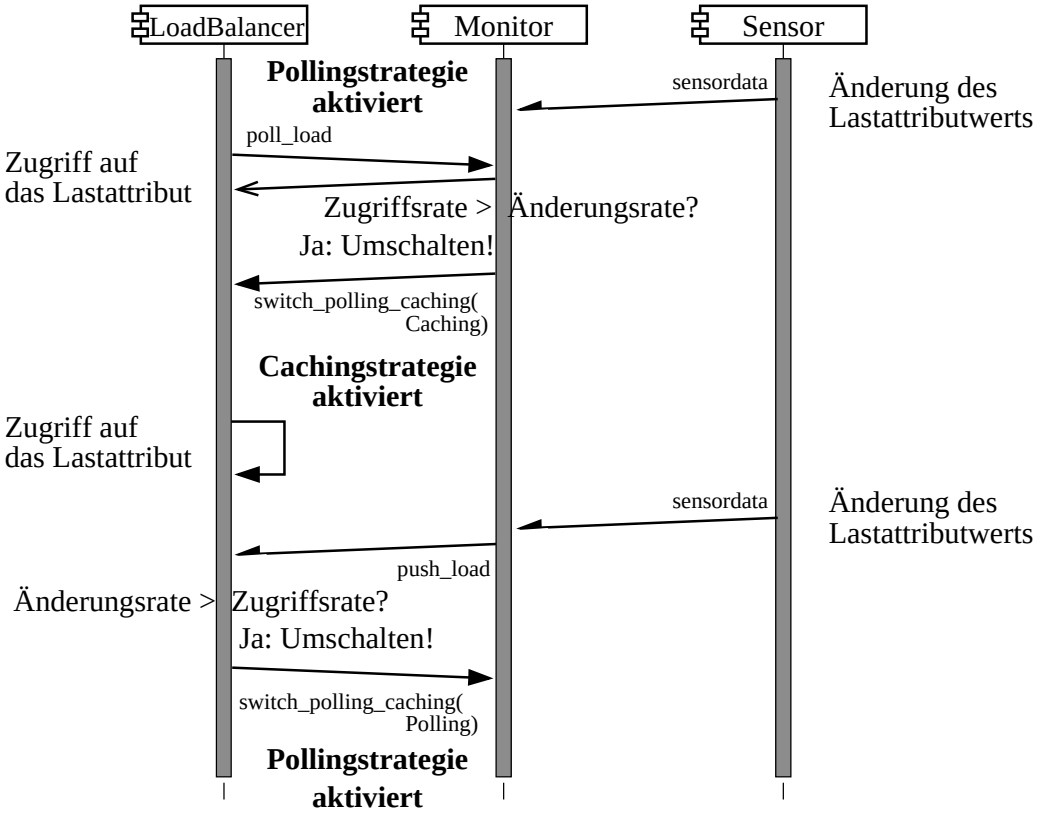

Abbildung 4.19: Dynamische Umschaltung der Aktualisierungsstrategie

Im Polling-Betrieb kann der Attributanbieter – in dieser Arbeit der Monitor – die Entscheidung, auf Caching umzuschalten, treffen. Dem Monitor liegen in diesem Fall die beiden benötigten Informationen vor. Die Änderungsrate ergibt sich aus dem zeitlichen Abstand, in dem der Sensor seine Daten an dem Monitor verschickt. Die Zugriffsrate kann aus dem zeitlichen Abstand, in dem die Polling-Anfragen des Load Balancers eingehen, bestimmt werden. Falls die Zugriffsrate größer als die Änderungsrate ist, informiert der Monitor den Load Balancer darüber. Dieser schaltet dann in den Caching-Betrieb um, der in dieser Situation optimal ist.
Im Caching-Betrieb kann der Monitor nicht mehr die Zugriffsrate ermitteln. Der Load Balancer kann jetzt aber aus den eintreffenden Cache-Updates Informationen über die Änderungsrate gewinnen. Falls die Änderungsrate größer als die Zugriffsrate des Load

Balancers ist, schaltet dieser auf den Polling-Betrieb um und teilt dies dem zuständigen Monitor mit.
Neben dem Aspekt der Netzlast kann bei den unterschiedlichen Aktualisierungsstrategien auch noch der Einfluß die Dienstvermittlungsdauer betrachtet werden. Gegenüber der Polling-Strategie, die erst bei der eigentlichen Dienstvermittlung aufgerufen wird, hat die Caching-Strategie den Vorteil, daß die benötigten Lastinformationen bereits bei der der Dienstvermittlung vorliegen. Eine Caching-Strategie verspricht daher eine kürzere Dienstvermittlungszeit.
Da die Abwägung zwischen einer evtl. die Netzlast reduzierenden Polling-Strategie und einer die Dienstvermittlungszeit verringernden Caching-Stratgeie subjektiv ist, können hierfür keine Automatismen angegeben werden. In der vorliegenden Implementierung kann kann daher beim Vergleich der Änderungs- und Zugriffsrate eine Gewichtung angegeben werden. Neben der angesprochenen subjektiven Präferenz kann dadurch auch berücksichtigt werden, daß die beim Caching eingesetzten Oneway-Operationsaufrufe wegen der fehlenden Rückantwort gegenüber den synchronen Polling-Anfragen weniger Netzlast verursachen.

4.2.3 Trader

4.2.3.1 Verwaltung der Diensttabellen

Die Verwaltung der Diensttabellen konnte von der bereits vorliegenden Traderimplementierung übernommen werden. Die strukturelle Assoziation der Kontext- und der Diensttabellen wurde bereits im Klassenstrukturdiagramm in Abbildung 4.13 erläutert.
Die ursprüngliche Implementierung der Abbildung der Baumstruktur auf die verwendete interne Datenstruktur war jedoch fehlerhaft und mußte korrigiert werden. Die Baumstruktur wurde durch eine LISP-ähnliche Tochter-Schwester-Struktur repräsentiert. Zur Traversierung eines Teilbaums wurde eine Rekursion eingesetzt, die versehentlich auch die Schwestern des Startknotens durchlief.[6]
Die Reihenfolge, in der die Einschränkung der Dienstangebotsmenge erfolgt, ist in Abbildung 4.20 zu sehen. Anhand des bei der Import-Anfrage angebenen Kontextes, durch den die Dienstangebotsmenge unter organisatorischen Gesichtspunkten strukturiert wird, wird im ersten Schritt der Startknoten des abzusuchenden Kontextteilbaums bestimmt. Jeder durchlaufene Knoten des Teilbaums enthält eine Diensttabelle, in der alle Dienstangebote des jeweiligen Kontextes eingetragen sind.
In nächsten Schritt werden für jede dieser Diensttabellen die Dienstangebote herausgesucht, die den *matchingConstraints* der Import-Anfrage entsprechen. Anhand der *matchingConstraints* kann ein logischer Ausdruck angegeben werden, den die Dienstattribute eines Dienstangebots erfüllen müssen, um in die Ergebnismenge aufgenommen zu werden.

[6] Der gleiche Fehler mußte auch in der Implementierung des Typmanagers, der die selbe interne Repräsentation der Baumstruktur verwendet, behoben werden.

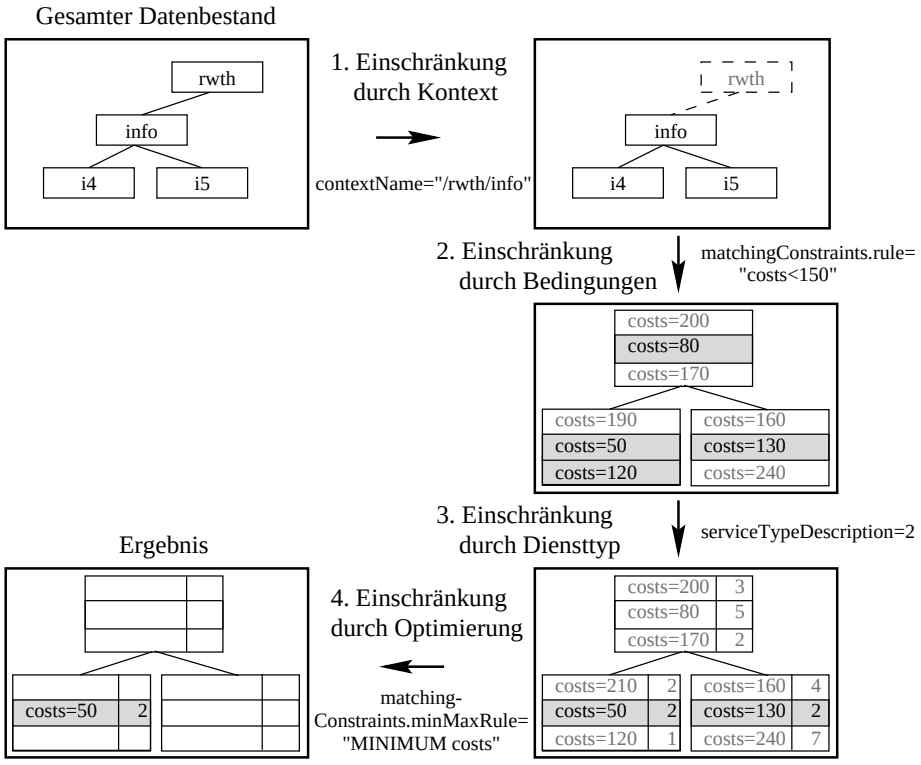

Abbildung 4.20: Dienstauswahl des Traders

Die Überprüfung, ob ein Dienstangebot überhaupt den gewünschten Diensttyp bereitstellt, erfolgt erst in Schritt 3. Die hierbei verwendeten numerischen Diensttypangaben stellen Indizes innerhalb des Typmanagers dar. Über diesen Typmanager ist es außerdem möglich, Untertyp-Beziehungen zwischen verschiedenen Diensttypen festzulegen.
In Schritt 4 kann eine Optimierung des bisherigen Ergebnisses durchgeführt werden. Hierzu kann die ausgewählte Dienstangebotsmenge bezüglich eines Attributs maximiert oder minimiert werden. Die Angabe der gewünschten Optimierung erfolgt im Import-Aufruf über das Strukturelement *matchingConstraints.minMaxRule*.
Zusätzlich zu der bereits vorhandenen Minimum- und Maximum-Optimierungsregel wurde eine Random-Auswahlwahlregel implementiert, die unabhängig von den Dienstattributwerten zufällig ein Element der Ergebnismenge aus Schritt 3 als Dienstangebot zurückliefert. Durch Verwendung der Random-Regel kann bereits eine einfache, statische Lastverteilung realisiert werden.

4.2.3.2 Zusammenarbeit mit dem Load Balancer

Die Zusammenarbeit der Traderkomponente mit der Load Balancer-Komponente erfolgt im wesentlichen innerhalb von Schritt 4. Im Anschluß an den dritten Schritt liegen die Kandidaten für die Lastbalancierung endgültig fest, so daß sie dem Load

Balancer mitgeteilt werden können. Da das weiter unten beschriebene Regelwerk einen Kompromiß zwischen Dienstattributoptimierung und Lastoptimierung finden soll, darf in Schritt 4 keine Optimierung bezüglich einer Diensteigenschaft durchgeführt werden. Stattdessen wird die Methode *Loadbalancer_i::insert_scoretable(...)* aufgerufen, um die Dienstangebote in die Liste der durch das Regelwerk zu bewertenden Server aufzunehmen. Der Quelltext der Methode *BalancedServiceTableItem::matchAgainstAll(...)*, in der die Schritte 2 bis 4 implementiert sind, ist in Abbildung 4.21 auszugsweise aufgeführt.

```
void BalancedServiceTableItem::matchAgainstAll(...){
  ...
  if (matchAgainstMatchingConstraint(matchingConstraint.rule)==true){
    if (matchAgainstType(servTypeDesc,idSeq)==true){
      if (use_loadbalancer_flag){
        // jedes gefundene Dienstangebot eintragen
        // 1. in die Dienstangebotsliste
        serviceOfferDetailList_var[++index]= ...
        // 2. In die Scoretable
        prop_name=
          matchingConstraint.minMaxRule.maxPropertyName();
        checkServicePropertyValues(servicePropertyValues,
                                   prop_name, prop_val);
        property_value=atoi(prop_val.value);
        Loadbalancer_Ptr->insert_scoretable(scoretable_Ptr,
         serviceOfferDetailList_var, index, property_value);
      }
      else // kein Loadbalancer benutzen: alte Semantik:
      {
        IndicatorType indicator=
         checkPropsAgainstMinMax(matchingConstraint.minMaxRule,
                                 actualMinMax);
        switch(indicator){
          case improves:
            serviceOfferDetailList_var->length(1);
            serviceOfferDetailList_var[index=0]= ...
          break;
          case emptyOrEquals:
            serviceOfferDetailList_var[++index]= ...
          break;
        };
      };
    };
  };
};
```

Abbildung 4.21: Einfügen eines passenden Dienstangebots in die Tabelle des Regelwerks

Nachdem der Trader seine Diensttabellen durchlaufen hat, steht das Dienstangebot fest und das Regelwerk des Load Balancer kann aufgerufen werden. Dies geschieht, wie

in Abbildung 4.22, die den Quelltext der Methode *Trader_i::importOffer2* enthält, zu sehen ist, durch den Aufruf von *Loadbalancer_i::get_best_scoreditem*.

```
void Trader_i::importOffer2(...){
  ...
  if (use_loadbalancer)
    my_scoretable=my_Loadbalancer->init_scoretable();
  // Eigentliche Tradingfunktion
  contextTable->searchContext( ... );
  // Angebot balancieren
  if (use_loadbalancer)
  {
    // Aus der Scoretable, in die der Trader parallel sein Angebot
    // eingetragen hat, den besten Eintrag suchen
    serviceofferindex=my_Loadbalancer->get_best_scoreditem(
      my_scoretable, minmax_flag, load_weight, property_weight, norm);
  };
  ...
};
```

Abbildung 4.22: Aufruf des Regelwerks durch den Trader

4.2.4 Load Balancer

4.2.4.1 Die globale Management Information Base

Neben der eigentlichen Lastverteilungsstrategie und dem Regelwerk zur Zusammenführung der Ergebnisse von Load Balancer und Trader wird eine globale Management Information Base benötigt, in der die übermittelten Lastinformationen der Server gespeichert werden. Die hierfür verwendete Klassenstruktur wurde bereits im Klassenstrukturdiagramm aus Abbildung 4.14 vorgestellt.
Neben der Speicherung muß sich die Klasse *ServerMonitorTable*, die die MIB verwaltet, auch um die Aktualisierung der Daten mittels Polling-Anfragen bzw. nach Cache-Updates kümmern. Dabei – aber auch während des Lastbalancierungsvorgangs – wird sehr oft auf diese Tabelle zugegriffen. Der Zugriff erfolgt über die Angabe des jeweils relevanten Servers. Die Identifikation von verteilten Objekten erfolgt in CORBA über eine interoperable Objekt-Referenz (IOR). Hierbei handelt es sich um eine circa 0,5 Kilobyte große Zeichenkette, die – über ASCII-Ziffern kodiert – den Namen der Serverimplementierung, dessen Schnittstellentyp und den zugehörigen Rechnernamen enthält. Da eine lineare Suche in der MIB-Tabelle sehr viele langwierige Zeichenkettenvergleiche benötigen würde, wurde der Zugriff auf die MIB über eine Hashtabelle beschleunigt.
Der CORBA-Standard sieht eine derartige Hashfunktion vor, allerdings kann die betreffende Methode nicht auf eine IOR, sondern nur auf ein instantiiertes CORBA-Objekt angewendet werden. Da für die in der MIB verwalteten Objekte jedoch nur

deren IOR vorliegt, mußte eine entsprechende Hashfunktion zusätzlich implementiert werden. Diese ist in Abbildung 4.23 dargestellt. Zur Generierung von möglichst eindeutigen Hashwerten wird ausgenutzt, daß eine IOR überwiegend aus ASCII-Ziffern besteht. Diese unterschieden sich lediglich in 4 Bits voneinander, weshalb die einzelnen Zeichen jeweils um 4 Bits versetzt zyklisch zu einem *long* aufaddiert werden.

```
unsigned long IORHash::hash(const char* ior, unsigned long max){
  unsigned int ior_length=strlen(ior);
  unsigned long carry,value=0;
  for (unsigned int i=0;i<ior_length;i++){
    value=(value<<4)+ior[i]; // 4 Bit-Shift und Zeichen dazuaddieren
    if((carry=value&0xf0000000)) // Wraparound notwendig?
    { // Ja: Wraparound simulieren
      value=value^(carry>>24); value=value^carry;
    }
  }
  return value%max;
};
```

Abbildung 4.23: Die verwendete Hashfunktion für CORBA-Objektreferenzen

4.2.4.2 Kommunikation mit den Monitoren

Bevor die Lastdaten in die MIB des Load Balancers eingetragen werden können, müssen sie von den Monitoren zum Load Balancer übertragen worden sein. In Abschnitt 4.2.2.2 wurden bereits die Caching-Strategie und ein Automatismus zur dynamischen Umschaltung zwischen Caching und Polling vorgestellt. Die Polling-Strategie wurde im Load Balancer in einer nebenläufigen und einer nicht-nebenläufigen Variante implementiert.
Die Ermittlung der Lastinformationen eines Servers durch einen Polling-Aufruf erfolgt im Anschluß an den Eintrag des zugehörigen Dienstangebots in die *ScoreTable*. In der nicht-nebenläufigen Variante blockiert der entfernte Operationsaufruf *monitor::poll_load(...)* daher die Dienstvermittlung des Traders bis die Antwort des Monitors vorliegt. Um bei einem Ausfall des Monitors oder einer langsamen Netzwerkverbindung dennoch einen zügigen Ablauf der Dienstvermittlung zu garantieren, wurde für die Polling-Anfragen ein gesondertes Timeout implementiert. Dazu wird die Zeit gemessen, die eine Polling-Anfrage im Durchschnitt benötigt. Das Timeout für einen Poll-Aufruf ergibt sich aus einem Vielfachen dieses Durchschnitts. Abbildung 4.24 zeigt einen Ausschnitt aus diesem Code, der die CORBA-Environments nutzt, um dort eine – von der Defaulteinstellung abweichendes – Timeout angeben zu können. Wird diese Zeit überschritten, so wird vom CORBA-Laufzeitsystem eine Exception erzeugt, in der eine Ausnahmebehandlung für den jeweiligen Aufruf durchgeführt werden kann.
Es hat sich gezeigt, daß die vorliegende Orbix-Implementierung bei kurzen Timeout-Vorgaben (im Bereich von wenigen Millisekunden) sporadisch eine Timeout-

```
void ServerMonitorTable::poll_load(const char* server)
{
  ...
  mytimeout=servermonitortableitem->get_polling_delay();
  CORBA::Environment env;
  env.timeout(2*mytimeout); // Timeout

  // Startzeit messen
  myClock.get_realtime(start_sec, start_usec);
  try{
    current_monitor_ptr->poll_load(server,
      item->loadItem->get_metric_needed(),load, env);
  }
  catch (CORBA::COMM_FAILURE &sysEx) {
    // Timeout
    env.clear(); // Exception entfernen
  }
  // Endzeit messen
  myClock.get_realtime(end_sec, end_usec);
  mytimeout=int((end_sec-start_sec)*1000+(end_usec-start_usec)/1000.0);
  // Antwortdauer merken
  servermonitortableitem->insert_polling_delay(mytimeout);
  ...
}
```

Abbildung 4.24: Timeoutbehandlung für die Polling-Anfrage

Exception erzeugt, obwohl die angegebene Zeitspanne noch gar nicht abgelaufen ist. Um diesen Fehler in Orbix zu umgehen, empfiehlt es sich daher, eine untere Schranke von ca. 1 Sekunde für die Timeout-Vorgabe einzusetzen.

Während des Wartens auf die Poll-Antwort könnte der Trader bereits mit der Dienstvermittlung fortfahren, da die beim Polling erfragten Lastinformationen erst zum Ende der Dienstvermittlung benötigt werden. Dies versucht sich die nebenläufige Variante der Polling-Strategie zu nutze zu machen. Die benötigte Nebenläufigkeit wurde durch Threads realisiert.

Ein Thread stellt einen „Prozeß im Prozeß“ dar, der unabhängig vom umfassenden Prozeß ausgeführt werden kann. Im zugrundeliegenden Betriebssystem Solaris 2 ist ein zweischichtiges Nebenläufigkeitsmodell implementiert [SG94, GGM93]. Abbildung 4.25 zeigt die dabei vorgenommene Unterscheidung zwischen Threads und leichtgewichtigen Prozessen. Die Threads stellen die Einheiten dar, mit denen der Programmierer umgeht, die leichtgewichtigen Prozesse werden vom Betriebssystem verwendet, um darin die Threads auszuführen. Nur die leichtgewichtigen Prozesse nehmen am CPU-Scheduling teil, während die Threads auf eine kooperative Zusammenarbeit angewiesen sind, in der sie sich einen leichtgewichtigen Prozeß teilen.

Ein Thread gibt „seinen“ leichtgewichtigen Prozeß ab, wenn er z.B. auf eine Semaphore wartet oder explizit die Funktion *thr_yield()* aufruft. Solaris 2 bemüht sich zwar, jedem Prozeß ausreichend viele leichtgewichtige Prozesse zur Verfügung zu stellen, den-

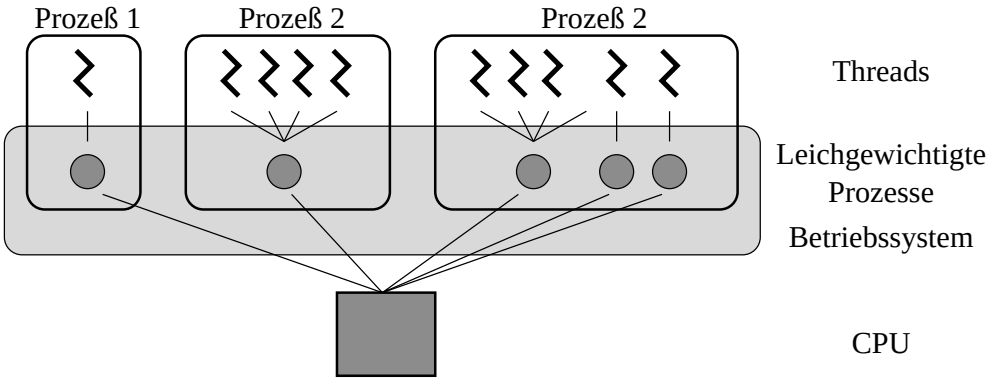

Abbildung 4.25: Das Thread-Modell in Solaris 2

noch zeigt die Praxis, daß dieses Bemühen nicht ausreichend ist. Daher ist es zusätzlich möglich, gebundene Threads zu erzeugen, die ihren „eigenen“ leichtgewichtigen Prozeß besitzen.
Der Quelltext in Abbildung 4.26 zeigt die Thread-Funktionen, die benötigt werden, um einen Orbix-Operationsaufruf nebenläufig abzusetzen.

```
...
thread_t my_pollthread; // Zeiger auf Pollthread
...
// Gebundenen Thread erzeugen
thr_create(NULL, 0, ServerMonitorTable::poll_load,
  (void*)threadstuff, THR_BOUND, my_pollthread)
// Dem neuen Thread die Chance geben, zu arbeiten
thr_yield();
...
```

Abbildung 4.26: Erzeugung eines gebundenen Threads

4.2.4.3 Lastverteilungsstrategien

Das dieser Implementierung zugrundeliegende Objektmodell sieht für die Speicherung und Bewertung von Lastinformationen eine abstrakte Basisklasse *loadstoreitem* vor, von der konkrete Implementierungen abgeleitet werden. Die entsprechende Klassenstruktur wurde bereits in Abschnitt 4.1.2.4 beschrieben.
Die in diesem Modell vorgesehene Lastbalancierung erfolgt zweistufig, da nach der Bewertung der Last auch noch der beste Kompromiß zwischen Trader- und Load Balancer-Ergebnis gefunden werden muß. Die eigentliche Auswahl eines Server geschieht erst im Regelwerk, das im nächsten Abschnitt beschrieben ist.
Die eingesetzten Lastverteilungsstrategien haben daher nur die Aufgabe, für das Regelwerk einen Bewertung der Last vorzunehmen. Für diese Arbeit wurden – neben der Traderauswahlstrategie *RANDOM* – drei verschiedene Lastverteilungsstrategien implementiert.

Diese basieren auf unterschiedlichen Lastmetriken. In Abschnitt 2.3.3 wurde bereits darauf hingewiesen, daß die Last, wie sie dort definiert ist, eine ungeeignete Basis zur Lastverteilung ist. Sie ist in zweierlei Hinsicht problematisch.
Zum einen berücksichtigt sie nur zurückliegende Vermittlungen. Aufträge, die zwar bereits an einen Server vermittelt, dort aber erst in Zukunft ausgeführt werden, weil sie noch in der Warteschlange stehen bzw. erst noch verschickt werden, haben keinen Einfluß auf die aktuelle Last. Eine auf der Last basierende Strategie würde in der Zwischenzeit beliebig viele weitere Aufträge an einen Server mit niedriger Last vermitteln, da es aufgrund der Mittelwertbildung eine Weile dauert, bis die neuen Aufträge die Durchschnittsbildung signifikant beeinflussen. Dies widerspricht der Forderung, möglichst aktuelle Lastinformationen zu verwenden.
Zum anderen gibt die Last eines Rechnerknotens keine Auskunft über dessen Leistungsfähigkeit. So ist in einem System mit inhomogener Leistungsfähigkeit ein ausgelasteter, aber schneller Rechner unter Umständen einem schwach ausgelasteten, aber langsamen Rechner vorzuziehen.
Unter diesen Gesichtspunken wurde darauf verzichtet, Strategien zu implementieren, die auf der Metrik der Last basieren. Stattdessen wurden Lastverteilungsstrategien implementiert, die ihre Entscheidung anhand der Anzahl der bisherigen Vermittlungen (Klasse *loadstoreitem_usage_count*), anhand der Warteschlangenlänge (Klasse *loadstoreitem_queuelength*), sowie anhand der geschätzten verbleibenden Antwortzeit (Klasse *loadstoreitem_estimated_time_to_work*) treffen.
Die beiden letzteren Metriken werden über die Monitore zur Verfügung gestellt. Die Anzahl der Vermittlungen kann innerhalb des Traders ohne Verwendung der Monitore ermittelt werden. Für die Messungen, die im nächsten Kapitel vorgestellt werden, wurde die Anzahl der Vermittlungen jeweils seit Beginn einer Messreihe betrachtet. In der Praxis bietet es sich an, hierfür ein Rückwärtsfenster zu verwenden, daß nur die Vermittlungen innerhalb eines bestimmten Zeitraums zählt.
Das im nächsten Abschnitt beschriebene Regelwerk zur Zusammenführung des Trader- und Load Balancer-Ergebnisses sieht vor, daß die Last jedes Servers über eine Punktzahl bewertet wird. Die implementierten Lastverteilungsstrategien verwenden eine Greedy-Heuristik [CLR94], die ihre Bewertungsentscheidung lediglich anhand der momentanen Serverlast trifft. Diese wird durch die jeweils verwendeten Metriken charakterisiert. Bei Verwendung der Metrik *ESTIMATED_TIME_TO_WORK* ist zusätzlich zu berücksichtigen, daß diese altert und dementsprechend von ihrem Wert das jeweilige Alter abgezogen werden muß.
Die in den drei Klassen implementierte Greedy-Strategie vergibt die beste Punktzahl an den Server mit dem geringsten Lastwert. In allen drei Klassen ergibt sich daher die Punktzahl unmittelbar aus dem gespeicherten Lastwert.

4.2.4.4 Regelwerk zur Ergebniszusammenführung

Zum Ende der Dienstvermittlung muß das Ergebnis des Traders, der sich nur mit den Diensteigenschaften befaßt, und das des Load Balancers, der nur die Last eines

Servers betrachtet, zusammengeführt werden. Hierfür wird ein Regelwerk eingesetzt, das abschließend aufgerufen wird, um einen Kompromiß zwischen möglichst guten Diensteigenschaften und einer möglichst niedrigen Last zu finden. Um eine derartige Abwägung vornehmen zu können, muß eine Bewertung dieser beiden unterschiedlichen Aspekte vorliegen. Dieser Vorgang ist in Abbildung 4.27 dargestellt.

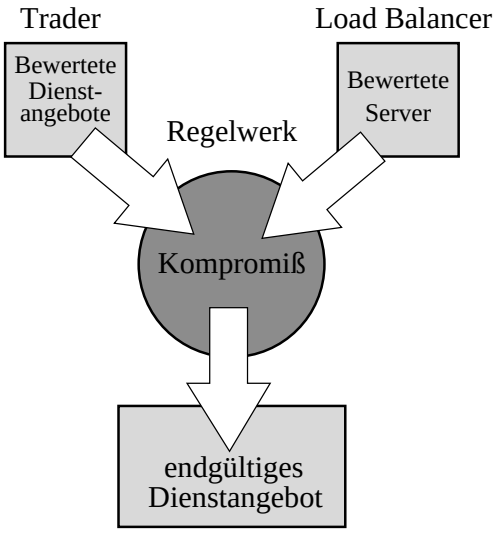

Abbildung 4.27: Die Zusammenführung der Ergebnisse von Trader und Load Balancer

Während der Load Balancer durch seine Lastverteilungsstrategie eine Punktzahl für die Auslastung des Dienstanbieters vergeben hat, ist für den Trader unklar, wie er die Diensteigenschaften eines Dienstangebots bewerten soll. Da die Integration der Lastverteilung transparent erfolgen soll, scheidet eine Erweiterung der Traderschnittstelle aus. Als Hinweis darauf, auf welcher Basis die Bewertung erfolgen soll, wird deshalb das Strukturelement *matchingConstraints.minMaxRule* der Trader-Importanfrage verwendet. Über dieses gibt ein Importer an, welches Dienstattribut minimiert oder maximiert werden soll. Es bietet sich daher an, den absoluten numerischen Wert dieses Dienstattributs als Bewertung anzusehen. Sollte ein Importer keinen Optimierungswunsch bezüglich eines Dienstattributs angegeben haben, so kann dies zum Anlaß genommen werden, eine reine Lastverteilung durchzuführen.
Zur Zusammenführung dieser beiden Bewertungen wird eine Abbildung f benötigt, welche die vorliegenden Bewertungen der Last B_{Last} und der Diensteigenschaft $B_{Diensteigenschaft}$ auf eine einzelne Bewertung der Kompromißgüte $B_{Kompromiß}$ abbildet:

$$f : B_{Last} \times B_{Diensteigenschaft} \mapsto B_{Kompromiß} \; .$$

Prinzipiell kommen hierfür beliebige Funktion $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ in Frage. Da die beiden Werte B_{Last} und $B_{Diensteigenschaft}$ als voneinander linear unabhängig anzusehen sind, bietet es sich an, sie als Komponenten eines zweidimensionalen Vektors aufzufassen. Unter der Voraussetzung, daß bei beiden Bewertungen Punktzahlen in der Nähe von

0 als eine gute Bewertung anzusehen sind, stellt der Nullvektor das bestmögliche Ergebnis dar. Die Länge des Vektors $\begin{pmatrix} B_{Last} \\ B_{Diensteigenschaft} \end{pmatrix}$ gibt unter diesen Umständen den Abstand zum Optimum an.
Bei der Bestimmung der Länge eines Vektors können verschiedene Normen verwendet werden [EJ91]. Die p-Norm eines Vektors X ist allgemein definiert als

$$\|X\|_p = \left(\sum_i x_i^p \right)^{\frac{1}{p}} .$$

Am gebräuchlichsten sind die Normen für $p = 1$ (Manhattandistanz) und $p = 2$ (euklidische Distanz), sowie für $\lim p \to \infty$ (Maximumsnorm).
In Abbildung 4.28 sind die Vektoren A, B und C eingezeichnet. An ihnen läßt sich die unterschiedliche Charakteristik der Normen darstellen. Die euklidische Norm und alle größeren Normen reagieren viel stärker auf einzelne Ausreißer in den Komponenten eines Vektors als die Manhattan-Norm. C hat deswegen bei Verwendung der euklidischen Distanz einen größeren Abstand vom Ursprung als A. Die Manhattandistanz berücksichtigt Ausreißer in den Komponenten nicht so stark, so daß hierfür die Punkte C und A den gleichen Abstand haben. Punkt A ist für beide Normen gleich weit vom Ursprung entfernt. Die euklidische Norm enstpricht dem „intuitivem“ Längenbegriff. Bei Verwendung dieser Norm haben die Vektoren A und B die gleiche Länge.

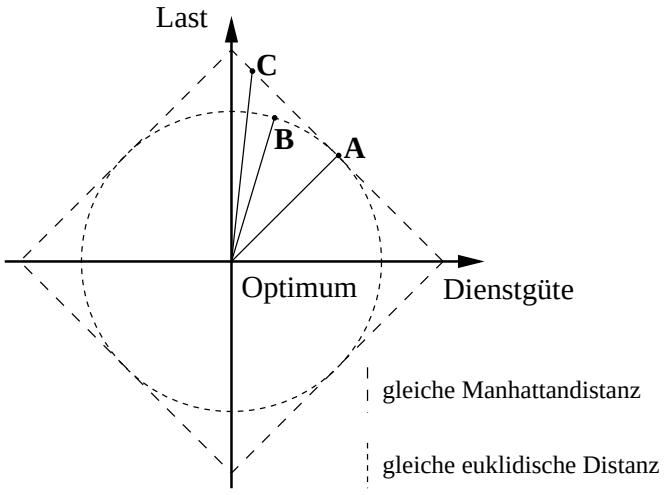

Abbildung 4.28: Euklidische Norm und Manhattannorm

Normen eignen sich demnach, um im Regelwerk einen Kompromiß zu bewerten, indem dessen Nähe zum Optimum betrachtet wird. Falls bei der Bewertung zusätzlich eine Präferenz bezüglich der Last oder des Dienstattributs berücksichtigt werden soll, können die einzelnen Komponenten mit einer Gewichtung w versehen werden. Im Falle der euklidischen Norm ergibt dies beispielsweise:

$$B_{Kompromiß} = \sqrt{(w_{Diensteigenschaft} \cdot B_{Diensteigenschaft})^2 + (w_{Last} \cdot B_{Last})^2} .$$

Diese Normen zur Abstandsbewertung lassen sich sehr einfach implementieren. Abweichend von den Defaulteinstellung für die zu verwendende Norm und die Gewichtungen können außerdem eigene Parameter hierfür beim Import über die gewünschten Dienstangebotseigenschaften (vgl. Abschnitt 4.1.2.5) angeben werden.
Um die Bewertung von Last und Dienstattribut vergleichen zu können, muß vorher eine Skalierung ihres Wertebereichs vorgenommen werden:

$$\tilde{B}_{Last_j} = \frac{B_{Last_j} - \min_i B_{Last_i}}{\max_i B_{Last_i} - \min_i B_{Last_i}}, \max_i B_{Last_i} \neq \min_i B_{Last_i} \ .$$

Diese Skalierung bildet den ursprünglichen Wertebereich der Lastbewertung auf das Intervall $[0..1]$ ab. Die Skalierung der Dienstattribute erfolgt analog, wobei zu berücksichtigen ist, ob der Attributwert minimiert oder maximiert werden soll.

```
ScoreTableItem* ScoreTable::calc_tablescores
     _and_return_minimal(int maximize_property_flag,
     float load_weight, float property_weight, float norm)
{
  ...
  for(p=head; p!=NULL; p=p->get_next())
  {
    my_load=p->get_load();
    my_property=p->get_property();
    // Skalieren der Werte bzgl. ihres Wertebereichs
    ...
    my_load=(my_load-min_load)/(max_load-min_load);
    my_property=(my_property-min_property)/(max_property-min_property);
    // Eigentliche Berechnung der Score
    // Hierzu wird eine n-Norm mit Gewichten verwendet
    p->set_score(pow(pow((load_weight)*(my_load), norm)
           +pow((property_weight)*(my_property), norm),
           1/norm));
    // Neue Score besser als bisher?
    if ((p->get_score()<minimal_score))
    {
      minimal_score=p->get_score();
      minimal_item=p;
    };
  };
  return minimal_item;
};
```

Abbildung 4.29: Quelltext des Regelwerks

Der Quelltext des implementierten Regelwerks, das die beschriebenen Schritte durchführt, ist in Abbildung 4.29 zu sehen. Als Ergebnis wird dabei das Dienstangebot zurückgeliefert, dessen Bewertungsvektor den kleinsten Abstand vom Optimum besitzt. In Abhängigkeit von den gewählten Gewichtungsfaktoren führt dies dann zu einer mehr oder weniger stark ausgeprägten Lastbalancierung.

Kapitel 5

Bewertung des Ansatzes

In diesem Kapitel erfolgt eine Bewertung des implementierten Ansatzes zur Dienstvermittlung unter Berücksichtigung der Last von Dienstanbietern. Zunächst werden die für die Bewertung herangezogenen Meßgrößen und Grundlagen aus der empirischen Statistik vorgestellt. Daran schließt sich die Beschreibung der verwendeten Testumgebung an. Die eigentlichen Meßergebnisse und deren Interpretation bilden den Hauptteil dieses Kapitels. Die dabei vorgenomme Bewertung betrachtet außer der Qualität der vorgenommenen Lastverteilung auch den Aufwand, der bei der Dienstvermittlung durch die Lastverteilungskomponente zusätzlich entsteht. Abschließend werden die Schlüsse, die bei der Interpretation der Meßergebnisse gezogen werden konnten, zusammengefaßt.

5.1 Meßgrößen

Zur Leistungsbewertung eines System werden Kriterien benötigt, anhand derer eine Beurteilung vorgenommen werden kann. Hierbei muß zwischen subjektiven und objektiven Kriterien unterschieden werden.
Aspekte, die die Dienstgüte betreffen, sind häufig subjektiv. Es lassen sich zwar Metriken verwenden, um die Dienstgüte zu bewerten. Diese werden jedoch in der gleichen Form bereits im zu bewertenden System eingesetzt, so daß die jeweiligen Metriken auf sich selbst angewandt würden. Es ist deshalb nicht möglich, eine objektive Empfehlung auszusprechen, welche Metrik im Regelwerk am besten eingesetzt werden sollte, um dort die Bewertung der Dienstgüte vorzunehmen. Ebenso ist die Wahl der Gewichtungsfaktoren für die Last bzw. die Dienstgüte letztendlich Geschmackssache, so daß diesbezüglich ebenfalls keine optimale Gewichtung bestimmt werden kann.
Zur Beurteilung der möglichen Lastmetriken und Lastverteilungsstrategien lassen sich jedoch vielfältige objektive Kriterien verwenden. Ebenso ist es möglich, den Overhead, der durch die Lastverteilung im Trading-Prozeß entsteht, zu messen. Als wichtigste Meßgröße kommt hierbei jeweils die mittlere Antwortzeit zum Einsatz. Die im einzelnen verwendeten Meßgrößen werden im folgenden vorgestellt.

5.1.1 Lastverteilungsgüte

Das Ziel der Lastverteilung besteht darin, systemweit die Antwortzeit zu minimieren. Als Kriterium zur Beurteilung der Qualität einer Lastverteilungsstrategie bietet es sich daher an, die über alle Dienstanbieter gemittelte Antwortzeit zu messen. Je näher die durch eine Lastverteilungsstrategie erzielte mittlere Antwortzeit an das theoretische Minimum rückt, desto besser ist diese Strategie. Als theoretisches Minimum kann die mittlere Wartezeit eines M/M/k-Warteschlangensystems angesehen werden, da in diesem Warteschlangenmodell von einer optimalen Lastverteilung ausgegangen wird. Für konstante Bedienzeiten beschreibt ein M/D/k-Modell mit seiner deterministischen Bedienrate das System noch exakter. Da das M/M/k-Modell den allgemeineren Fall beschreibt, wird im weiteren auf diese Modellierung zurückgegriffen.
Als weiteres Kriterium zur Bewertung einer Lastverteilungsstrategie wird die Last, die bei jedem einzelnen Dienstanbieter entsteht, betrachtet. Eine gute Lastbalancierung sorgt dafür, daß die Last bei allen Anbietern gleich groß ist. Für ein inhomogenes System mit unterschiedlich leistungsfähigen Rechnern kann es bei einer sehr niedrigen Systemauslastung allerdings auch sinnvoll sein, schnelle Rechner stärker zu belasten.
Die durchschnittliche Anzahl der Aufträge im System, d.h. in der Wartschlange und im Server, bietet ein Bewertungskriterium, das ähnlich wie die Last interpretiert werden kann. Gegenüber der Last fließt hier zusätzlich noch die Warteschlangenlänge ein.
Als letzte Meßgröße zur Bewertung einer Lastverteilungsstrategie wird die Anzahl der Dienstnutzungen verwendet. Bei homogenen System sollten alle Server gleich oft aufgerufen worden sein, im inhomogenen Fall sollte sich hier das Verhältnis der Leistungsfähigkeit der einzelnen Rechnerknoten widerspiegeln.

5.1.2 Lastverteilungsaufwand

Neben der Güte der Lastverteilungsstrategien soll auch der Einfluß der Lastverteilung auf die Dienstvermittlung untersucht werden. Hierbei interessiert besonders, um wieviel die Dienstvermittlungszeit durch die Betrachtung der Last steigt. Als Bewertungskriterium wird hierfür die mittlere Antwortzeit des Traders verwendet.
Da die Lastverteilungskomponente nicht nur Einfluß auf die Dienstvermittlungsdauer hat, wird außerdem noch die Netzlast betrachtet, die durch die Übermittlung der Lastinformationen von den Monitoren zum Load Balancer entsteht. Dies geschieht, indem die Anzahl der hierdurch entstehenden Kommunikationsvorgänge gezählt wird.

5.1.3 Fehler der Meßgrößen

Die gemessenen Werte stellen nur Stichproben dar. Aus diesem Grunde können lediglich Schätzungen für die wahren Werte der jeweiligen Meßgrößen vorgenommen werden. Die empirische Statistik bietet Methoden, um Schätzwerte und deren Fehler zu bestimmen [Stö93]. Die für die nachfolgenden Bewertungen eingesetzten Methoden sollen an dieser Stelle kurz vorgestellt werden.

Aus n gemessenen Werten x_i kann über das arithmetische Mittel eine Schätzung W_n für den wahren Wert W vorgenommen werden:

$$W_n = \overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \ .$$

Ein solcher Schätzwert enthält keine Aussage darüber, wie gut diese Schätzung ist. Es muß zusätzlich die Streuung der Meßwerte x_i betrachtet werden, um die Qualität eines Schätzwertes beurteilen zu können. Ein einfaches Maß für die Streuung der Meßwerte ist die empirische Varianz σ_n^2, die sich aus der quadratischen Abweichung der einzelnen Meßwerte vom Schätzwert berechnet:

$$\sigma_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2 \ .$$

Als weiteres Maß für die Güte einer Schätzung können Konfidenzintervalle verwendet werden. Der wahre Wert W befindet sich mit Wahrscheinlichkeit α außerhalb des zugehörigen $(1-\alpha)$-Konfidenzintervalls, das den Schätzwert umschließt. Das $(1-\alpha)$-Konfidenzintervall ist wie folgt definiert:

$$W_n - \frac{\sigma_n}{\sqrt{n}} \cdot t_{T;1-\frac{\alpha}{2};n-1} \leq W \leq W_n + \frac{\sigma_n}{\sqrt{n}} \cdot t_{T;1-\frac{\alpha}{2};n-1} \ .$$

Bei $t_{T;\alpha;n}$ handelt es sich um das Perzentil der t- bzw. Student-Verteilung. Ein Perzentil gibt den Wert an, den ein Anteil α aller Werte unterschreitet bzw. ein Anteil von $(1-\alpha)$ überschreitet. Für $\alpha = 0.5$ ergibt sich beispielsweise der Median.
Bei den nachfolgenden Diagrammen dieses Kapitels ist zur Beurteilung der Güte der Messungen jeweils das 90%-Konfidenzintervall in Form von Fehlerbalken angegeben.

5.2 Testumgebung

Im folgenden wird eine Beschreibung der Testumgebung, in der die Messungen stattfanden, gegeben.

5.2.1 Lasterzeugung

Die Umgebung, in der die Messungen durchgeführt werden, soll einerseits den Einsatz des zu bewertenden Systems in der Praxis widerspiegeln, andererseits soll die Umgebung ein genau definiertes Umfeld darstellen, um die Messungen nachvollziehbar und wiederholbar zu gestalten.
Es wurde daher versucht, ein Szenario zu schaffen, daß deterministische Anfragen an den Trader enthält und von der resultierenden Servernutzung her einem realen Einsatz möglichst nahe kommt. Da keine konkreten Untersuchungen aus der Praxis bekannt sind, die die Anfragecharakteristik in einem Client-Server-System beschreiben, wurde

eine synthetische Anfragesequenz erstellt, die geeignet erscheint, ein reales Anfrageverhalten zu simulieren.
Bei der Erzeugung einer solchen Anfragesequenz müssen verschiedene Dinge beachtet werden. Zunächst einmal soll jede Sequenz eine bestimmte Last erzeugen, so daß Messungen für verschiedene Lasten durchgeführt werden können. Insbesondere muß darauf geachtet werden, daß das Gesamtsystem nicht überlastet wird. Hierzu muß ein stabiler Zustand vorliegen, d.h. es dürfen nicht mehr Anfragen gestellt werden, als das System überhaupt in der gegebenen Zeit bearbeiten kann [Bol89]. Dennoch sollen zeitlich beschränkte Überlastsituationen auftreten, um das Verhalten der Lastverteilungsstrategien in temporären Überlastsituationen beurteilen zu können. Dies kann erreicht werden, indem neben kurzzeitigen Anfrage-Bursts auch entsprechende Ruhephasen in der Anfragesequenz enthalten sind.
Zur Generierung einer Anfragesequenz mit einer derartigen Charakteristik wurde ein Pseudozufallszahlengenerator verwendet, der in einem bestimmten Zeitraum eine bestimmte Anzahl von Anfragen plaziert. Bei n Servern läßt sich für A Anfragen die Gesamtdauer T einer Sequenz, die die Last ρ erzeugt, aus den einzelnen Bedienzeiten t_{b_i} berechnen:

$$T = \frac{A}{\rho} \frac{1}{\sum_{i=1}^{n} \frac{1}{t_{b_i}}} \; .$$

Falls verschiedene Klassen von Anfragen mit jeweils unterschiedlicher Bediendauer zum Einsatz kommen, ergibt sich die Gesamtdauer T_Σ aus der Summe der Sequenzdauern für die einzelnen Anfrageklassen:

$$T_\Sigma = \sum_{\forall \text{Klassen } C_j} T_j \; .$$

Für die nachfolgenden Messungen wurde eine Sequenz mit 100 Anfragen verwendet. Die generierte Sequenz, die bei allen Messungen zum Einsatz kam, ist in Abbildung 5.1 dargestellt. Für die unterschiedlichen zu generierenden Lasten und unterschiedlichen Leistungsstärken der eingesetzten Rechner wurde diese Sequenz entsprechend skaliert. Bei den Messungen kamen Sequenzdauern zwischen 32 und 1046 Sekunden zum Einsatz. Die Bedienzeiten lagen zwischen 1.1 und 16.0 Sekunden.
Die eigentliche Last wird von Servern erzeugt, die auf Anfrage Rechenzeit verbrauchen und somit CPU-Last verursachen.

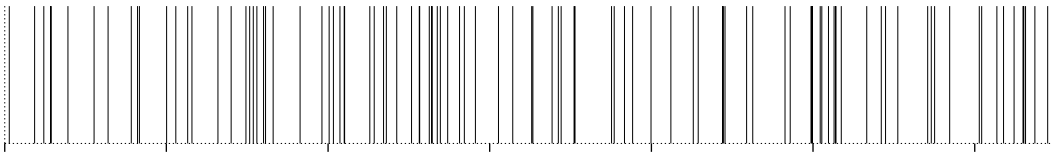

Abbildung 5.1: Die für die Messungen verwendete Anfragesequenz

5.2.2 Meßwertaufnahme

Zur Aufnahme der Meßwerte wurde ein zentraler Client verwendet, der entsprechend der generierten Anfragesequenz Import-Anfragen an den Trader stellt und direkt im Anschluß den vom Trader vermittelten Server nutzt. Da während der temporären Überlastsituationen der Client auf mehrere ausstehende Anfragen warten muß, werden Threads verwendet, um währenddessen weiterhin neue Anfragen abschicken zu können.

Die zu bewertenden Meßgrößen fallen an verschiedenen Stellen des Systems an. Die mittlere Antwortzeit aller Server kann im Client gemessen werden. Dort wird ebenfalls die Dienstvermittlungsdauer, d.h. die Antwortzeit des Traders gemessen. Der zusätzliche Kommunikationsaufwand der Lastverteilung wird im Trader erfaßt, da bei diesem alle relevanten Lastnachrichten eintreffen. Serverspezifische Daten, wie dessen Last, dessen mittlere Warteschlangenlänge sowie dessen Anzahl der Nutzungen werden von den jeweiligen Monitoren gemessen. Hierfür können zum Teil die bereits genutzten Filterpunkte verwendet werden, zum Teil müssen aber auch separate Meßpunkte installiert werden. Zur Zeitmessung wird, wie bereits in Kapitel 4 beschrieben, die Unix-Betriebssystemfunktion *gettimeofday* verwendet.

Die Messungen wurden in einem lokalen Netzwerk (10/100 Mb/s-Ethernet) durchgeführt. Die verwendete Rechnerhardware ist in Tabelle 5.1 aufgeführt. Die Messungen fanden jeweils mit vier Servern, die den gleichen Dienst exportierten, statt. Für die Messungen eines Systems mit homogener Rechenleistung wurden die Server auf den Rechnern *Aoxomoxoa, Hauptquartier, Intensivstation* und *Voltaire* gestartet. Das inhomogene System wurde durch die Rechner *Ostgote, Titanic, Voltaire* und den durch einen Hintergrundprozeß auf 50%-Rechenleistung gedrosselten Rechner *Aoxomoxoa* realisiert. Die einzelnen Rechner waren ansonsten während der Messungen weitgehend ungenutzt. Der Trader lief auf dem Rechner *Starfighter*. Bei der Bewertung der Dienstvermittlungsdauer wurde zur Aktualisierung der Lastinformationen eine Caching-Strategie verwendet.

Rechnername	Aoxomoxoa, Hauptquartier, Intensivstation, Voltaire	Starfighter,Titanic	Ostgote
Hersteller	Sun	Sun	Sun
Modell	Ultra 1 Creator (3D)	Ultra 1 140	SPARCStation 5
Anzahl CPUs	1	1	1
Taktrate	167 Mhz	143 Mhz	110 Mhz
Hauptspeicher	128MB	64MB	32MB
Virtueller Speicher	450MB	176MB	101MB

Tabelle 5.1: Kenndaten der verwendeten Rechnerknoten

5.3 Meßergebnisse

5.3.1 Lastverteilungsgüte

In diesem Abschnitt wird allein die Güte der verwendeten Lastverteilungsstrategien betrachtet. Die Import-Anfrage beschränkte sich auf die Angabe des gewünschten Diensttyps. Eine Einschränkung bezüglich der Dienstattribute fand nicht statt. Der Einfluß von Attributen bzw. die Leistungsfähigkeit des Regelwerks, das einen Kompromiß zwischen Dienstgüte und Last trifft, wird in Abschnitt 5.3.2 untersucht.

5.3.1.1 Reine Lastbalancierung bei homogener Serverleistung

Die verschiedenen Lastverteilungsstrategien sind für unterschiedlichen Situationen unterschiedlich gut geeignet. Zunächst wird auf den homogenen Fall eingangen, bei dem alle vier verwendeten Rechner die gleiche Rechenleistung besitzen. Die Anfragen haben alle die gleiche Bedienzeit t_b=1.1s.
Abbildung 5.2 stellt für diesen Fall die mittlere Antwortzeit dar. Neben den implementierten Strategien *Random*, *Usage_Count*, *Queuelength* und *Estimated_Time_To_Work*, die auf den im vorhergehenden Kapitel beschriebenen Metriken beruhen, sind als theoretische Grenzen die Antwortzeiten der M/M/1 und M/M/4-Warteschlangenmodelle eingezeichnet.

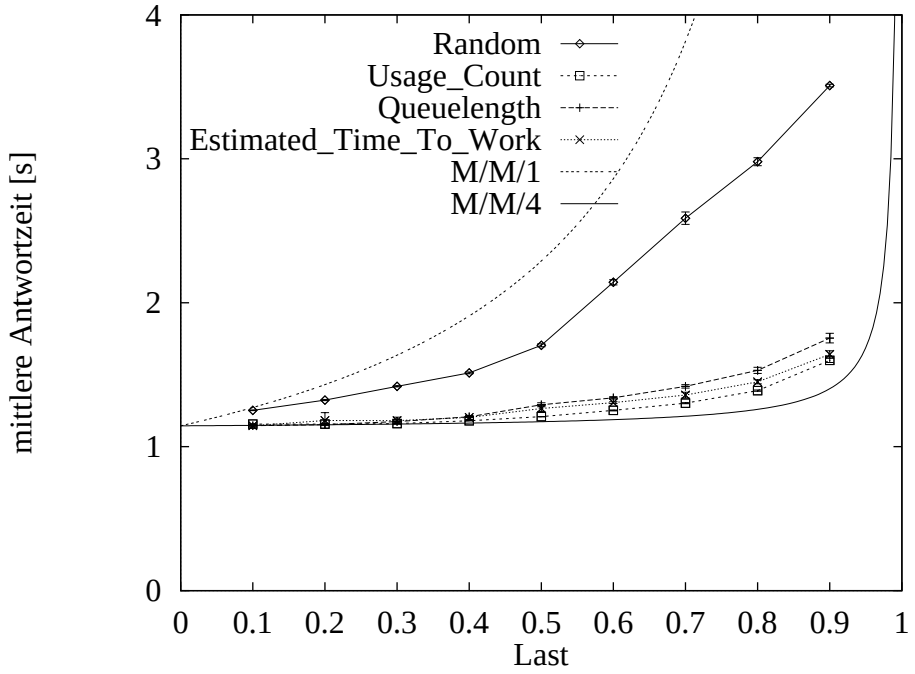

Abbildung 5.2: Antwortzeiten für reine Lastbalancierung (homogenes System)

Die Antwortzeiten der drei Strategien *Usage_Count*, *Queuelength* und *Estimated_Time_To_Work* liegen relativ dicht beieinander und kommen dem theoretischen Optimum noch nahe. Die zufällige Verteilung der Anfragen auf die Server schneidet hingegen schlecht ab. Dies ist darauf zurückzuführen, daß der verwendete Pseudozufallsgenerator keine sehr gute Gleichverteilung besitzt. Für einen Zahlengenerator, der auch über einen kurzen Zeitraum eine Gleichverteilung realisert, wäre ein ähnlich gutes Verhalten wie bei *Usage_Count* zu erwarten. Die Charakteristik eines derartigen Zahlengenerators wäre dann allerdings auch nicht mehr als zufällig zu bezeichnen.
Als beste Strategie stellt sich *Usage_Count* heraus, bei der der bisher am wenigsten genutzte Server vermittelt wird. Da jeweils die gleichen vier Server vermittelt werden, entspricht dies der von einigen Tradern angebotenen Strategie *Cyclic_Choice*, bei der mehrere in Frage kommende Dienstanbieter reihum vermittelt werden.
Für den Fall, daß die Rechenleistung homogen ist, reicht daher das alleinige Wissen des Traders aus, um eine gute Lastverteilung zu realieren. Dies gilt allerdings nur für den Fall, daß die Server nicht ohne die Vermittlung des Traders genutzt werden und keine Hintergrundlast durch weitere Server verursacht wird. Der weiter unten beschriebenen inhomogene Fall legt den Schluß nahe, daß diese Strategie versagt, sobald Server ohne das Wissen des Traders genutzt werden.
In diesem Fall bietet es sich an, auf *Queuelength* oder *Estimated_Time_To_Work* auszuweichen. Bei Lasten oberhalb von ρ=0.4 schneidet die Verteilung anhand der geschätzten verbleibenden Bearbeitungszeit etwas besser ab als die alleinige Betrachtung der Warteschlangenlänge.
Die Güte der Messungen ist recht hoch, so daß die Fehlerbalken der 90%-Konfidenzintervalle größtenteils in den Markierungen der Meßwerte untergehen. Lediglich in den Fällen, in denen an den verwendeten Rechnern gleichzeitig gearbeitet und dadurch unregelmäßige Hintergrundlast erzeugt wurde, fällt die Qualität der Messungen etwas schlechter aus.
Bei Betrachtung der einzelnen Server läßt sich die Charaktistik der einzelnen Lastverteilungsstrategien beurteilen. Abbildung 5.3 gibt die Last an den einzelnen Servern *Aoxomoxoa, Hauptquartier, Intensivstation* und *Voltaire* wieder. Es wurden hierfür exemplarisch die Gesamtsystemlasten ρ=0.3, 0.5, 0.7, 0.9 herausgegriffen.
Es ist zu sehen, daß die Verteilung, die durch die dynamischen Strategien *Queuelength* und *Estimated_Time_To_Work* erzielt wird, mit steigender Last gleichmäßiger wird. Bei kleiner Systemauslastung ergibt sich eine Rangfolge bezüglich der Auslastung der Server. Dies ist dadurch zu erklären, daß bei kleinen Lasten nur wenige Aufträge gleichzeitig zu bearbeiten sind. Falls sich z.B. alle Server im Leerlauf befinden, weisen die beiden dynamischen Strategien einen neuen Auftrag immer demjenigen Server zu, der als erster vom Trader gefunden wurde. Die statische *Random*-Strategie bzw. die *Usage_Count*-Strategie aus dem Grenzfeld zwischen statischen und dynamischen Strategien besitzen hingegen keine Präferenzen. Bei den dynamischen Strategien läßt sich bei niedrigen Lasten aus der Anzahl der Anfragen pro Server die Reihenfolge ablesen, in der die Server im Trader gespeichert sind. Abbildung 5.4 verdeutlicht dies. Obwohl bei den Strategien *Queuelength* und *Estimated_Time_To_Work* bei niedriger

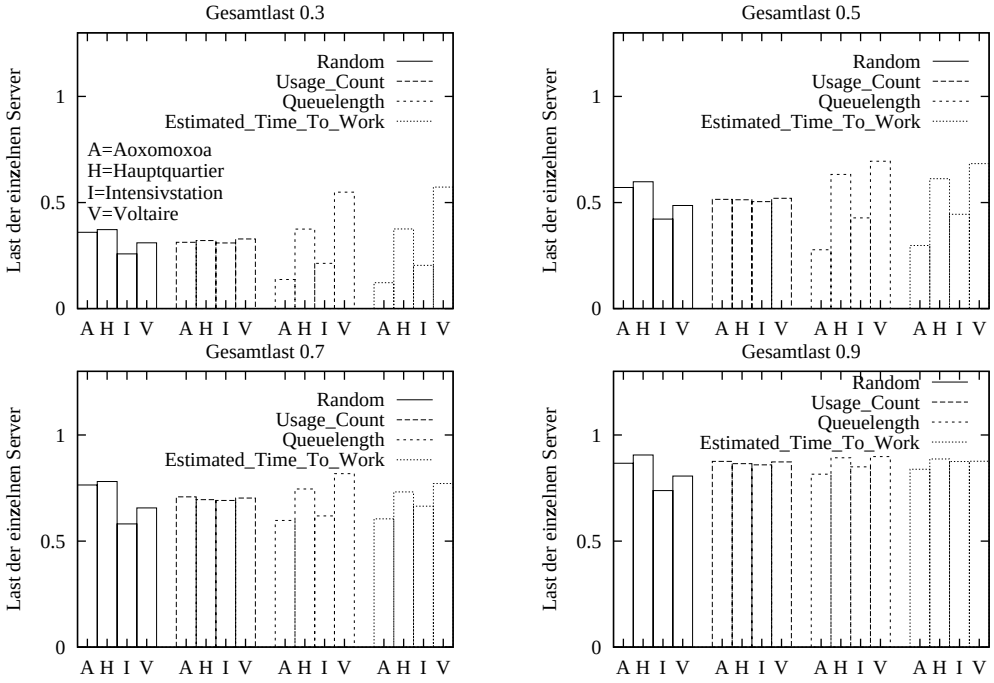

Abbildung 5.3: Last an den einzelnen Servern (homogenes System)

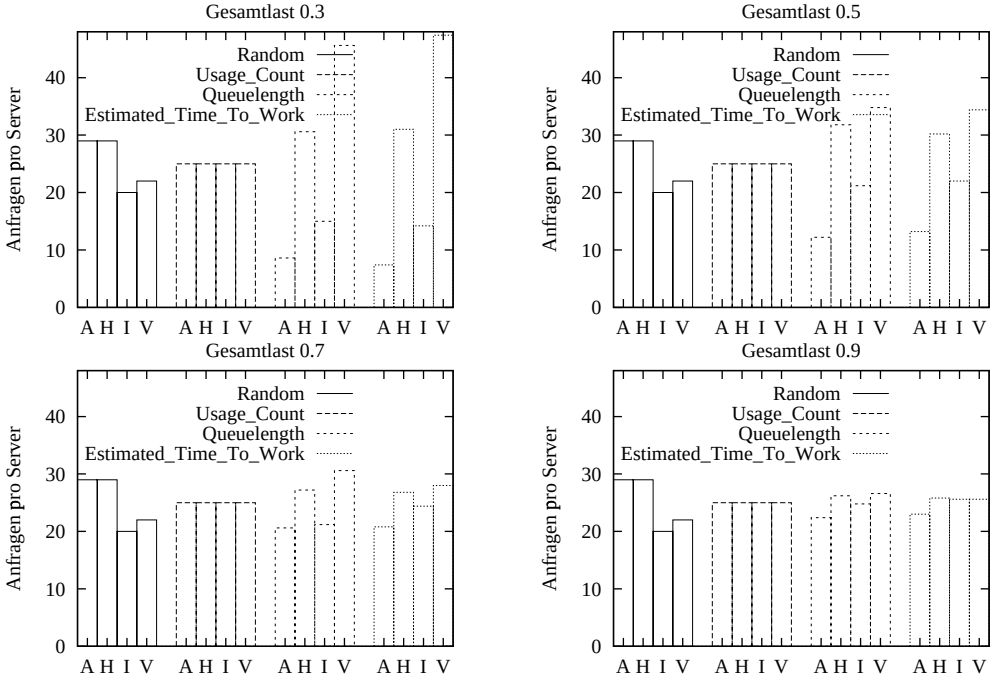

Abbildung 5.4: Anfragen pro Server (homogenes System)

Last die Verteilung der Anfragen ungleichmäßig ist, bedeutet dies jedoch nicht, daß dies auch zu schlechten Antwortzeiten führt. Die Ungleichverteilung tritt nur dann

auf, wenn die Server ungenutzt sind. Bei steigender Last steigt auch die Anzahl der genutzten Server, so daß sich bei hohen Lasten dementsprechend eine Gleichverteilung ergibt. Falls auch bei niedrigen Lasten eine gleichmäßige Lastverteilung gewünscht ist, bietet es sich an, die beiden dynamischen Strategien mit den beiden statischen Strategien zu koppeln.

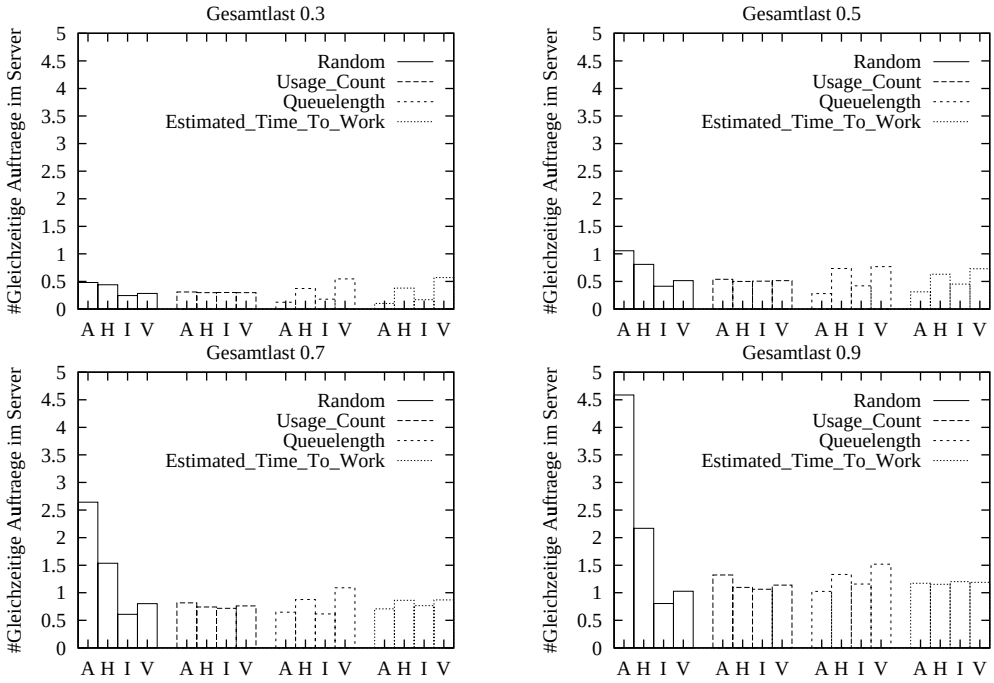

Abbildung 5.5: Aufträge im System pro Server (homogenes System)

In Abbildung 5.5 ist abschließend die Anzahl der gleichzeitigen Aufträge pro Serversystem dargestellt. Bei kleinen Lasten wartet selten ein Auftrag in der Warteschlange. Hierfür ergibt sich daher ungefähr die Last der einzelnen Server, wie sie bereits in Abbildung 5.3 zu sehen war. Bei großen Lasten machen sich hingegen die wartenden Aufträge bemerkbar. Dementsprechend steigt dort die Anzahl der gleichzeitigen Aufträge in den jeweiligen Serversystemen. Bemerkenswert ist, daß die Strategie *Estimated_Time_To_Work* bei hohen Lasten eine gleichmäßigere Verteilung der Aufträge im Serversystem sowohl bezüglich des insgesamt besser abschneidende *Usage_Count*-Verfahren als auch des – genau diese Anzahl betrachtenden – *Queuelength*-Verfahren erzielt.

5.3.1.2 Reine Lastbalancierung bei inhomogener Serverleistung

Im inhomogenen Fall zeigen die betrachteten Lastverteilungsstrategien ein anderes Verhalten. Die Bedienzeit der verwendeten Anfrage beträgt für das im folgenden untersuchte Szenario je nach Rechner 1.1s (*Voltaire*), 1.3s (*Titanic*), 1.7s (*Ostgote*) bzw. 2.2s (*Aoxomoxoa* gedrosselt).

Die hieraus resultierenden mittleren Antwortzeiten sind in Abbildung 5.6 in Abhängigkeit von der Last aufgetragen. Als theoretische Obergrenze ist die mittlere Antwortzeit für eine M/M/1-Warteschlange mit der Bedienzeit des langsamsten Rechners eingezeichnet. Dies entspricht einer Lastverteilungsstrategie, die immer den langsamsten Rechner auswählt. Als Untergrenze wurde eine M/M/4-Warteschlange verwendet, die sich für den Fall ergibt, daß viermal der schnellste Rechner zur Verfügung steht. Da tatsächlich auch langsamere Rechner genutzt werden, entspricht dies nicht der wirklichen theoretischen Untergrenze, die speziell bei hohen Lasten etwas höher liegen dürfte. Da eine exakte analytische Betrachtung aber an dieser Stelle zu komplex ist, wurde diese Vereinfachung getroffen.

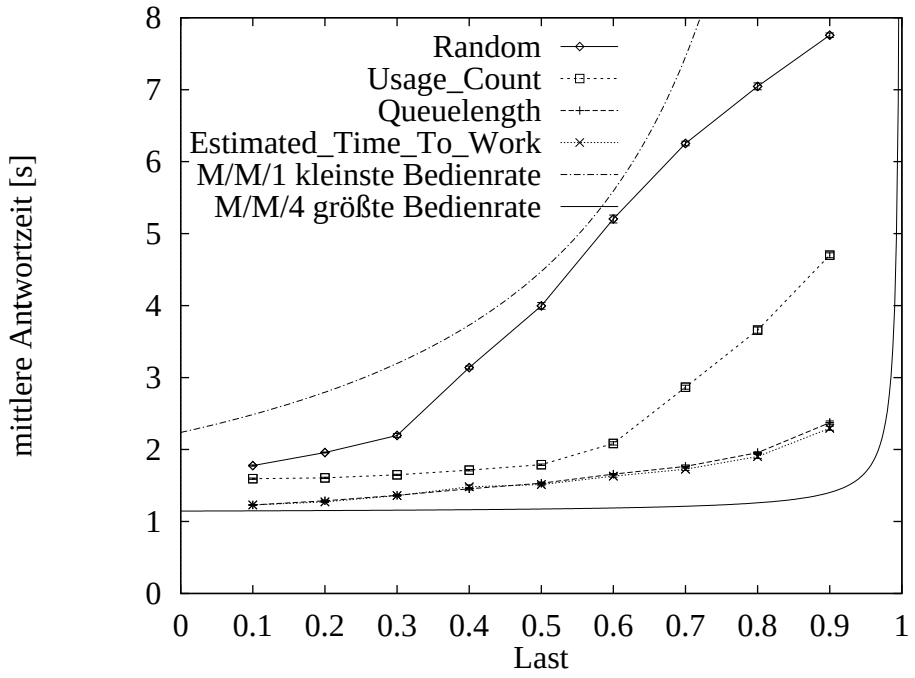

Abbildung 5.6: Antwortzeiten für reine Lastbalancierung (inhomogenes System)

Auch im inhomogenen Fall erweist sich die zufällige Verteilung als schlechteste Strategie. Im Gegensatz zum homogenen Fall schneidet die Verteilung nach der Anzahl der Dienstnutzungen (*Usage_Count*) nun ebenfalls deutlich schlechter ab als die beiden dynamischen Strategien. Dieses Verhalten entspricht auch der Erwartung, da leistungsschwache Rechner weiterhin genausoviele Anfragen zugewiesen bekommen wie leistungsstärkere.
Die beiden dynamischen Lastverteilungsstrategien können sehr viel besser auf die inhomogene Serverleistung eingehen. Die durch sie resultierende mittlere Antwortzeit verläuft immer noch vergleichsweise nahe an der eingezeichneten, vereinfachten Untergrenze. Obwohl die Strategie, die auf der Metrik *Estimated_Time_To_Work* beruht, die unterschiedlichen Bediendauern der einzelnen Server erfaßt, erweist sich diese Strategie nur minimal der einfacheren *Queuelength*-Strategie überlegen.

Zur Visualisierung der Charakteristik der einzelnen Strategien sind in den Abbildungen 5.7 bis 5.9 die Meßgrößen nach Servern (*Aoxomoxoa, Ostgote, Titanic, Voltaire*) aufgeschlüsselt. Als Lastwerte wurde diesmal exemplarisch ρ=0.3, 0.7, 0.8 und 0.9 herausgegriffen. Im Gegensatz zum homogenen Fall fällt bei den Messungen mit inhomogenen Rechnern die Qualität dieser Meßwerte schlechter aus. Da die Konfidenz allerdings nur bei den dynamischen Verfahren schlechter wurde, kann vermutet werden, daß es sich dabei nicht um Fehler in der Messung handelt. Vielmehr dürfte der Load Balancer selber jeweils bei Entscheidungen, die „auf der Kippe“ standen, in verschiedenen Meßdurchläufen jeweils zu unterschiedlichen Ergebnissen gekommen sein, weil die entsprechenden Monitordaten minimal anders ausgefallen sind. Dieses Phänomen zeigt sich auch bei den Szenarien der noch folgenden Abschnitte.

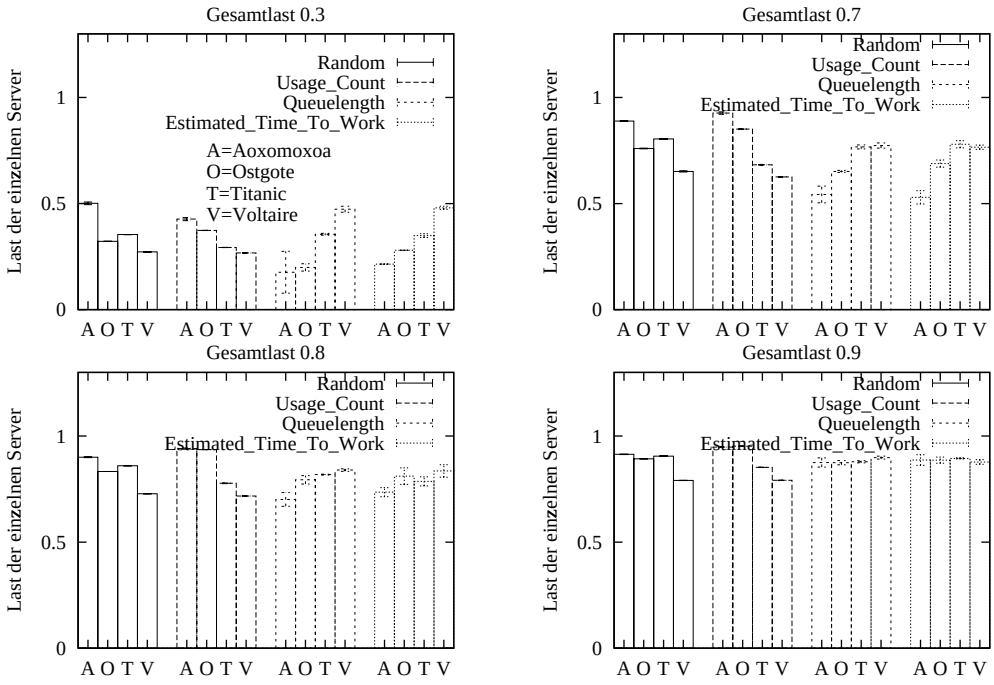

Abbildung 5.7: Last an den einzelnen Servern (inhomogenes System)

Die Betrachtung der Last an den einzelnen Servern bestätigt, wie schlecht statische Verfahren im inhomogenen Fall die Last verteilen. Die *Usage_Count*-Strategie vergibt gleichmäßig Aufträge an die unterschiedlich leistungsfähigen Server. Dies hat zur Folge, daß bei schwachen Rechnern schnell eine hohe Last erreicht ist, während die schnelleren Rechner noch freie Kapazitäten besitzen.
Die dynamischen Verfahren hingegen bevorzugen schnellere Server (*Voltaire, Titanic*), so daß diesen mehr Aufträge zugewiesen werden und deren Auslastung höher als die der langsameren Rechner ist. Generell ist festzustellen, daß eine derartige Bevorzugung nur bei niedrigen Lasten möglich ist, da bei hoher Last die gesamte zur Verfügung stehende Rechenleistung benötigt wird, so daß auch an schwächere Rechner Aufträge verteilt werden.

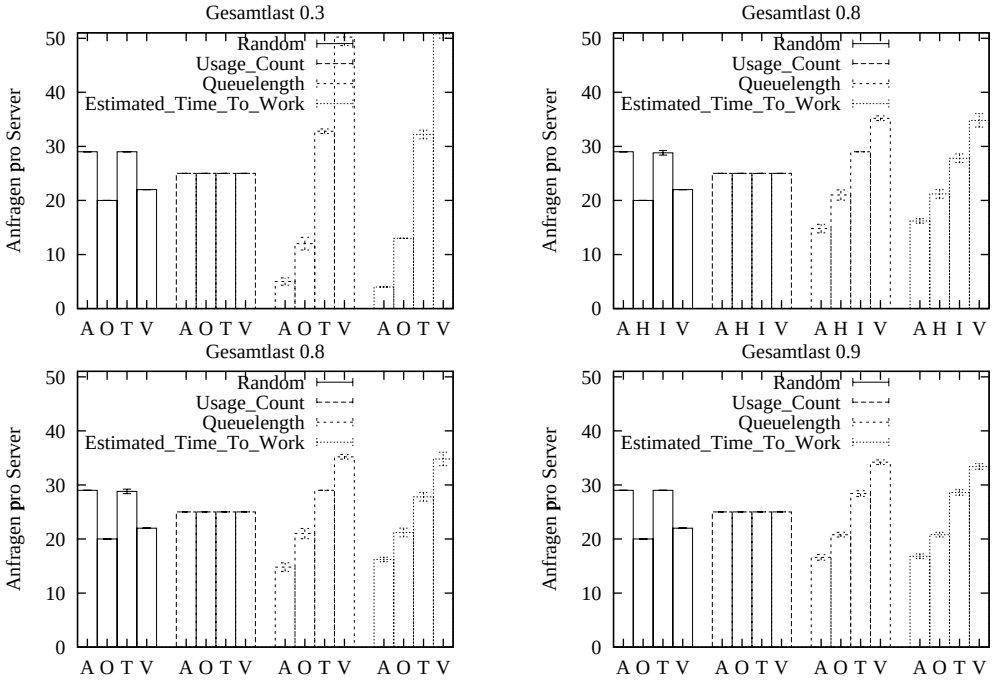

Abbildung 5.8: Anfragen pro Server (inhomogenes System)

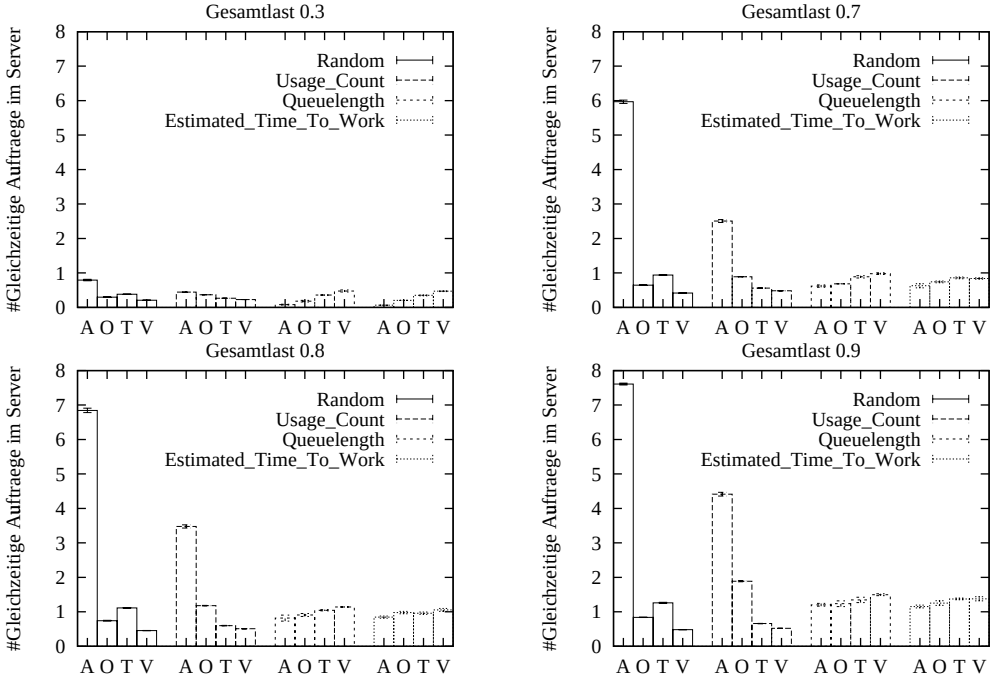

Abbildung 5.9: Aufträge im System pro Server (inhomogenes System)

Bezüglich der Aufträge, die sich gleichzeitig im Server-Warteschlangensystem befinden, spiegelt sich bei niedriger Auslastung – wie zu erwarten – die Last wieder. Bei

steigender Last ergibt sich bei den statischen Verfahren eine extreme Ungleichverteilung, da sich an den langsamen Servern Aufträge stauen. Die dynamischen Verfahren sorgen hingegen für eine annähernde Gleichverteilung, da den Servern, die ihre Warteschlange schneller leeren, immer entsprechend neue Aufträge zugeteilt werden.
Abschließend läßt sich feststellen, daß dynamische Lastverteilungsstrategien bei inhomogenen Systemen den statischen Strategien überlegen sind. Wissen über die Ausführungszeiten der einzelnen Aufträge bringt gegenüber einer rein auf der Warteschlangenlänge basierenden dynamischen Strategie keine herausragenden Vorteile. Um die Qualität der statischen Verfahren zu erhöhen, wäre es denkbar, deren Zuteilung entsprechend der Leistung der einzelnen Server zu gewichten. Damit dies möglich ist, muß allerdings die Leistungsfähigkeit aller Server bekannt sein. Diese Voraussetzung ist in einem offenen Dienstmarkt nicht gegeben. Durch die dort mögliche transparente Servermigration kann sich die Leistung eines Dienstanbieters unbemerkt verändern. Darüberhinaus kann ein statisches Verfahren prinzipiell keine wechselnde Hintergrundlast erfassen.

5.3.1.3 Reine Lastbalancierung bei homogener Serverleistung mit verschiedenen Anfrageklassen

Während bei den vorhergehenden Szenarien immer die gleiche Anfrage an die Dienstanbieter gestellt wurde, werden nun vier verschiedene Klassen von Anfragen eingesetzt. Diese besitzen unterschiedliche Bedienzeiten.
Dieses Szenario kann so interpretiert werden, daß auf den Serverrechnerknoten eine wechselnde Hintergrundlast für eine unregelmäßige Verlängerung der Dienstbearbeitungszeit sorgt.
Zunächst soll der Fall mit homogener Rechnerleistung untersucht werden. Die Bediendauer der einzelnen Anfrageklassen lag hiefür bei 8.0s, 3.0s, 1.1s und 0.4s. Die verschickten Anfragen wurden zufällig aus den vier Klassen gewählt, wobei aus jeder Klasse die gleiche Anzahl von Anfragen gezogen wurde.
Die über alle Anfrageklassen gemittelte Antwortzeit ist in Abbildung 5.10 dargestellt. Der Versuch, dieses Szenario analytisch über Warteschlangenmodelle zu erfassen, wurde hier nicht unternommen. Bei der Berechnung der mittleren Antwortzeiten wurde über die Antwortzeiten aller Anfragenklassen gemittelt. Weil die zu untersuchenden Strategien nicht das Ziel haben, eine Shortest-Job-First o.ä. Strategie zu implementieren, wurde das Bedienverhalten für einzelne Klassen nicht untersucht.
Trotz der unterschiedlichen Bedienzeiten der Klassen reicht es, die gemittelte Antwortzeit zu betrachten. Zwar ergibt sich die Antwortzeit aus der Wartezeit und der Bedienzeit, die von Klasse zu Klasse variiert, dennoch muß dies nicht berücksichtigt werden. Die Anzahl der Anfragen pro Klasse ist für jede Meßreihe gleich, so daß in die gemittelte Antwortzeit aufgrund der Linearität des arithmetischen Mittelwertoperators lediglich eine konstante Summe, die sich aus der mittleren Bedienzeit ergibt, eingeht.
Im Vergleich mit dem entsprechenden klassenlosen homogenen Fall fällt zunächst auf, daß der Abstand der drei Strategien *Usage_Count, Queuelength* und *Estima-*

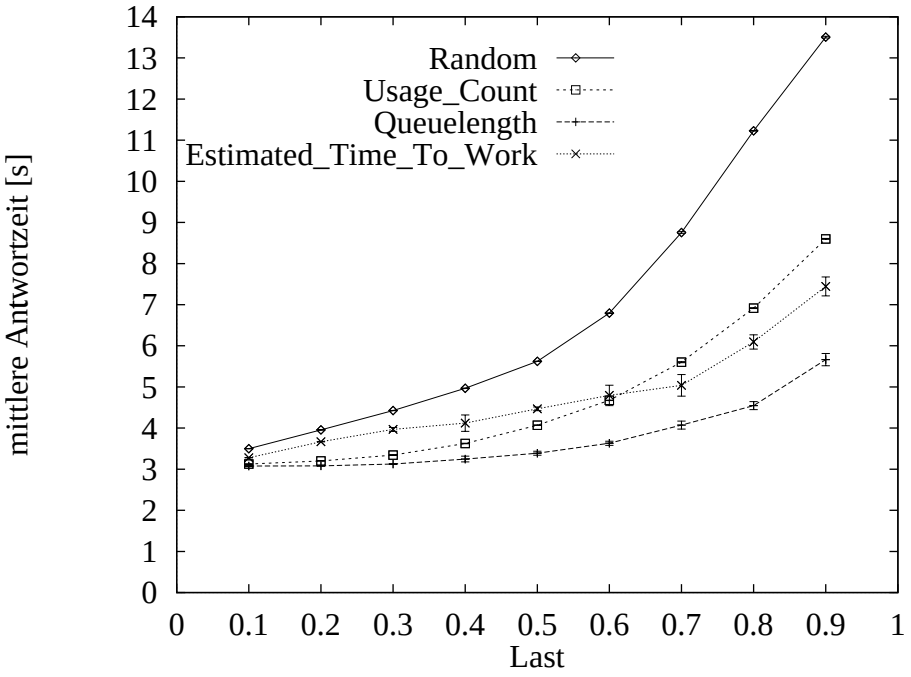

Abbildung 5.10: Antwortzeiten für reine Lastbalancierung (homogenes System mit Anfrageklassen)

ted_Time_To_Work voneinander größer geworden ist. Bei näherer Betrachtung zeigt sich sogar, daß sich deren Verhalten umgekehrt hat.

Abgesehen von der *Random*-Strategie, die erneut am schlechtesten abschneidet, ist in diesem Fall die gleichmäßige Verteilung per *Usage_Count* bei hohen Lasten die schlechteste Lastverteilungsstrategie. Dies ist dadurch zu erklären, daß die eingehenden Anfragen zwar gleichmäßig auf die einzelnen Server verteilt werden, die Klasse – und damit die Größe – der eingehenden Anfrage ist jedoch zufällig. Die einzelnen Server sind daher mit unterschiedlich langen Anfragen beschäftigt.

Bei einer Systemlast, die kleiner als ρ=0.6 ist, erweist sich allerdings die Strategie *Estimated_Time_To_Work* als schlechter. Dies ist dadurch zu erklären, daß sie zur Schätzung der verbleibenden Bearbeitungszeit einen gleitenden Mittelwert aus den vergangenen Anfragebediendauern bildet. Da die Bediendauern aufgrund der unterschiedlichen Klassen variieren, wird eine falsche Schätzung vorgenommen. Bei steigender Last wird der Schätzfehler von der Tatsache, daß diese Strategie gegenüber *Usage_Count* dynamisch ist, aufgewogen.

Als beste Strategie stellt sich *Queuelength* heraus. Bei ihrer Betrachtung der Warteschlangenlänge besitzt diese Strategie kein Wissen über die Größe der darin enthaltenen Aufträge, stattdessen werden alle Aufträge als gleichwertig angesehen. Die Verwendung von keinem Wissen bezüglich der Auftragsgröße erweist sich als besser als die Verwendung von falschem Wissen, wie es bei *Estimated_Time_To_Work* der Fall ist.

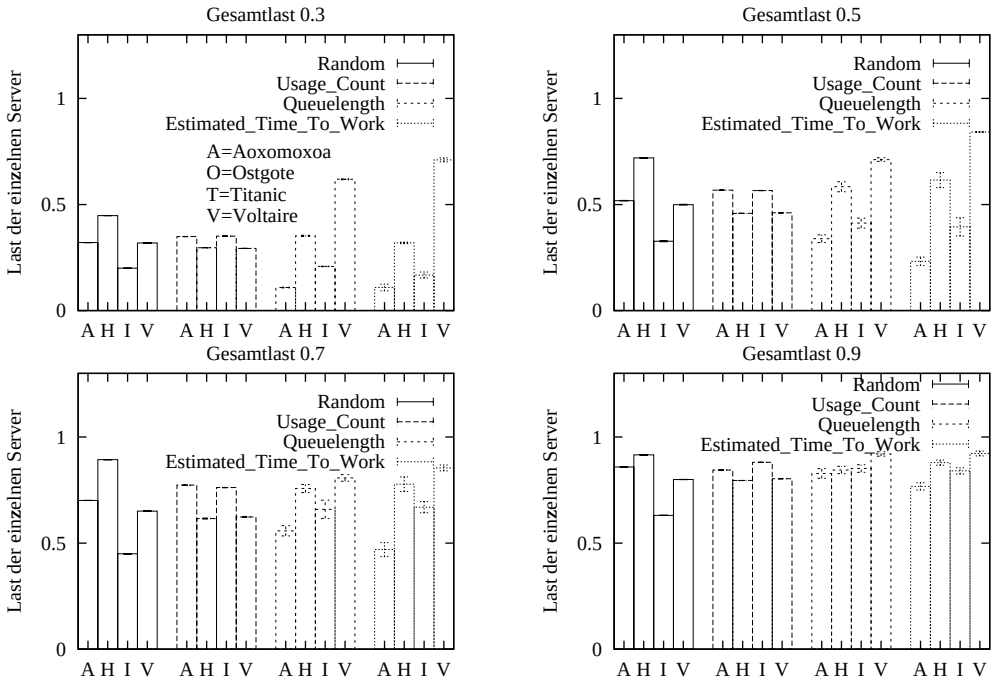

Abbildung 5.11: Last an den einzelnen Servern (homogenes System mit Anfrageklassen)

Die Meßwerte für die einzelnen Server sind in den Abbildungen 5.11 bis 5.13 dargestellt. Bezüglich der dynamischen Strategien ist – wie schon beim klassenlosen homogenen Fall – festzustellen, daß bei kleinen Lasten die Server ihrer Reihenfolge in der Traderdiensttabelle entsprechend genutzt werden. Mit steigender Gesamtlast verschwindet auch in diesem Fall diese Präferenz zugunsten einer gleichmäßigen Verteilung.
Bei hohen Lasten erzielt die Strategie *Queuelength* die gleichmäßigste Verteilung der Last auf alle Server. Dieses Ergebnis bestätigt das gute Abschneiden, das diese Strategie bezüglich der mittleren Antwortzeit aufweist. Alle anderen Verfahren scheitern beim Versuch, im Hochlastbereich eine Gleichverteilung der Last zu erreichen.
Die Bewertung der Strategien anhand der Überprüfung, ob die Anfragen gleichmäßig über die Server verteilt wurden, ist im vorliegenden Szenario nicht zulässig. Wegen der unterschiedlichen Anfrageklassen kann die gleiche Anzahl von Aufträgen zu einer unterschiedlichen Last führen. Dies zeigt die Betrachtung der Strategie *Usage_Count*. Obwohl die Anzahl der Anfragen für jeden Server gleich ist, ergibt sich bezüglich der auftretenden Warteschlangenlängen eine sehr ungleichmäßige Verteilung.
Für die Last ρ=0.9 fällt in Abbildung 5.13 auf, daß die Strategie *Estimated_Time_To_Work* scheinbar eine sehr gute Verteilung der Aufträge über die Server bzw. deren Warteschlangen realisiert. Gleichzeitig fällt aber auch auf, daß dort die Konfidenzintervalle sehr groß sind, weshalb eine derartige Aussage nicht getroffen werden kann. Ein Vergleich mit Abbildung 5.10 weist auf ein eher schlechtes Ver-

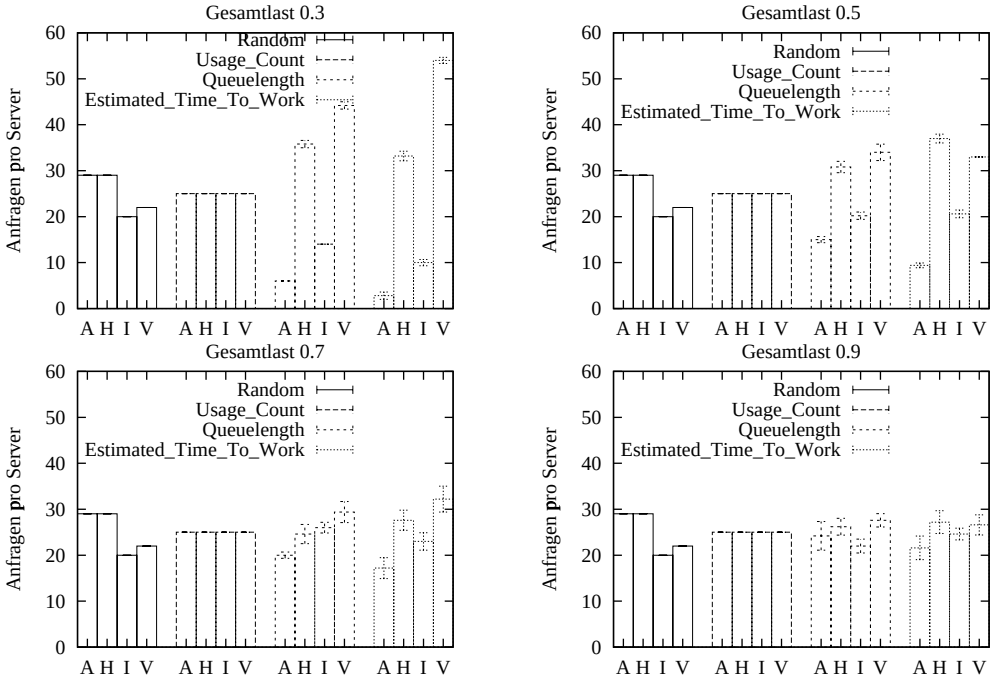

Abbildung 5.12: Anfragen pro Server (homogenes System mit Anfrageklassen)

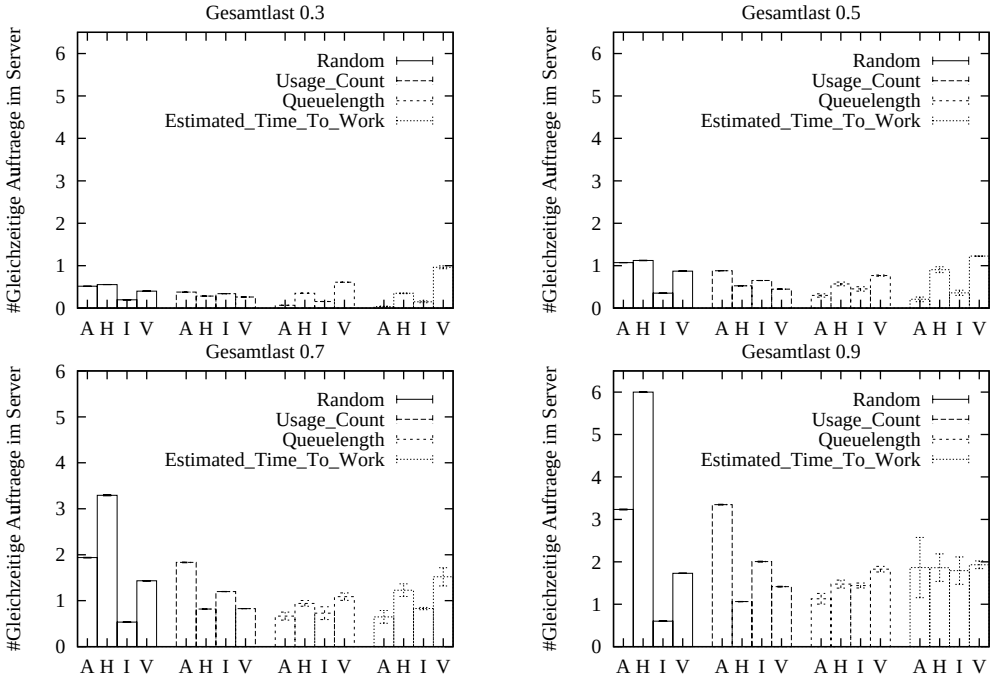

Abbildung 5.13: Aufträge im System pro Server (homogenes System mit Anfrageklassen)

halten hin und legt nahe, daß die Schätzung der verbleibenden Antwortzeit wegen der unterschiedlichen Bedienzeiten instabil ist und für eine entsprechende Varianz der Meßwerte sorgt.
Es läßt sich festhalten, daß die Strategie *Queuelength* am besten mit den unterschiedlich großen Anfragen dieses Szenarios umgehen kann. Die Strategie *Usage_Count* ist für hohe Lasten schlecht geeignet. *Estimated_Time_To_Work* schneidet zwar bei niedrigen Lasten schlechter ab, dafür verhält sie sich bei hohen Lasten besser. Ihre Leistungsfähigkeit ist dort aber deutlich schlechter als die der *Queuelength*-Strategie. Die probabilistische Verteilung stellt sich wie bei allen anderen Szenarien als ungeeignet heraus.

5.3.1.4 Reine Lastbalancierung bei inhomogener Serverleistung mit verschiedenen Anfrageklassen

Als letztes Szenario soll eine Kombination der beiden vorhergehenden Szenarien untersucht werden. Neben mehreren Klassen mit unterschiedlichen Bedienzeiten wurde das inhomogene System, das bereits im vorletzten Abschnitt beschrieben wurde, eingesetzt. Es wurden dieselben Anfrageklassen wie in Abschnitt 5.3.1.3 verwendet. Durch die inhomogene Rechenleistung variieren die resultierenden Bedienzeiten einer Klasse von Dienstanbieter zu Dienstanbieter.
Die für diesen Fall gemessenen Antwortzeiten sind in Abbildung 5.14 zu sehen.

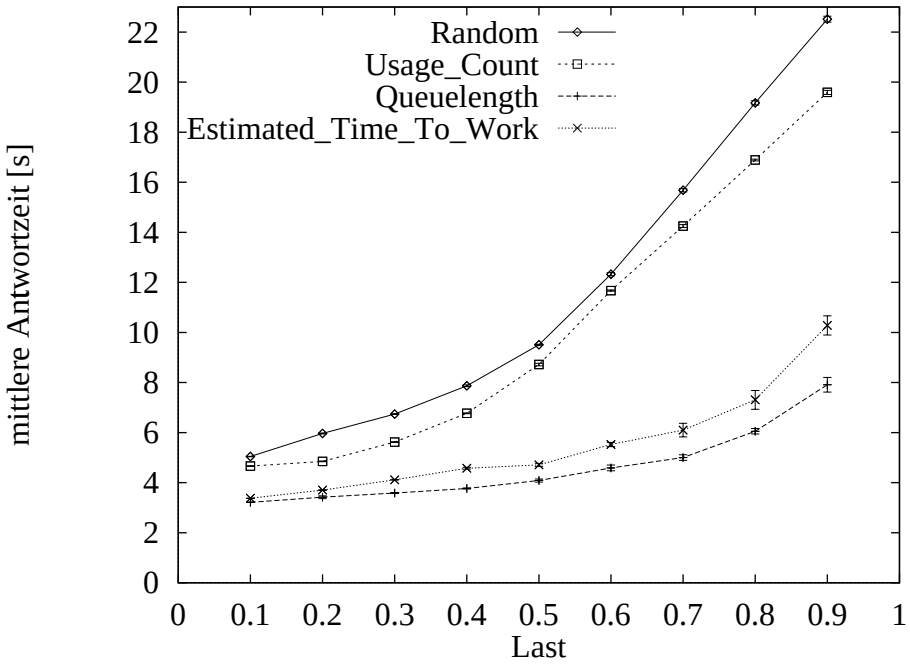

Abbildung 5.14: Antwortzeiten für reine Lastbalancierung (inhomogenes System mit Anfrageklassen)

Dieses doppelt inhomogene Szenario scheidet endgültig die dynamischen von den statischen Lastverteilungsverfahren. Die in den vorhergehenden Fällen immer noch ganz gut abschneidende *Usage_Count*-Strategie degeneriert nun von der Leistungsfähigkeit her annähernd zur *Random*-Strategie. Die beiden dynamischen Strategien sind auch in diesem Fall in der Lage, sich der inhomogenen Umgebung anzupassen.
Bezüglich der Verfahren *Queuelength* und *Estimated_Time_To_Work* ergibt sich im wesentlichen das gleiche Bild wie bereits beim homogenen Fall mit verschiedenen Anfrageklassen. Die rein auf der Warteschlangenlänge basierende Strategie läßt sich nicht von den unterschiedlichen Bediendauern der verschiedenen Klassen beeinflussen. *Queuelength* erreicht daher geringere Antwortzeiten als *Estimated_Time_To_Work*.

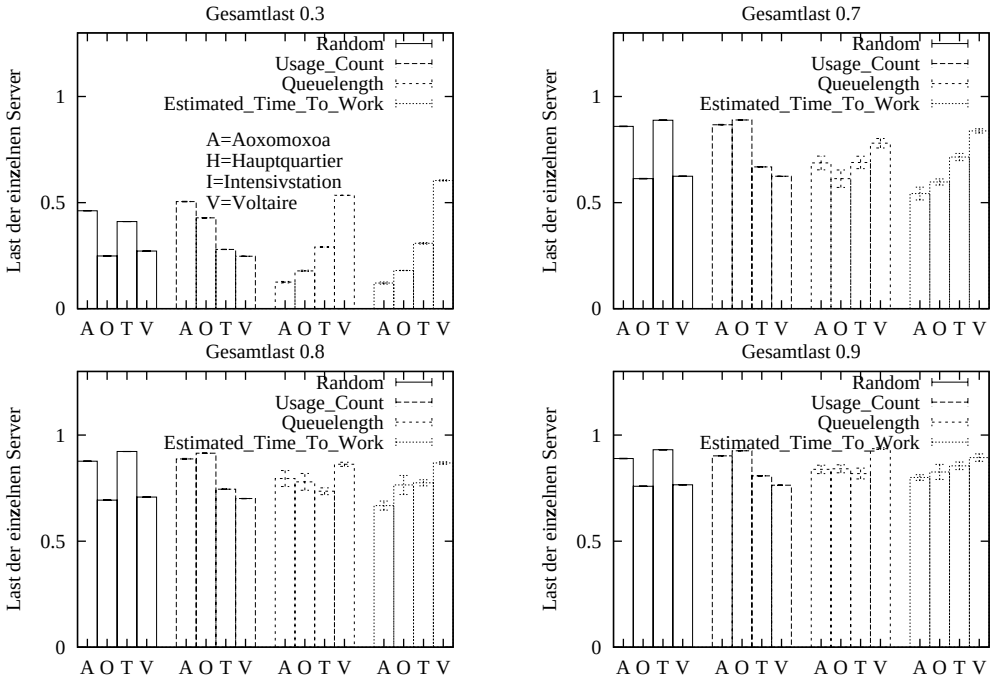

Abbildung 5.15: Last an den einzelnen Servern (inhomogenes System mit Anfrageklassen)

Die Last ist in Abbildung 5.15 nach den einzelnen Servern aufgeschlüsselt. Bei niedrigen Lasten zeigt sich wieder das von den implementierten dynamischen Verfahren gewohnte Verhalten, Server entsprechend der Reihenfolge des Auffindens bevorzugt zu vermitteln. Die Überlegenheit der dynamischen Strategien bestätigt sich bei hohen Lasten, da hier eine ausgeglichenere Lastverteilung erreicht wird als bei den statischen Verfahren. Es fällt jedoch auf, daß selbst für ρ=0.9 sowohl von *Queuelength*, als auch von *Estimated_Time_To_Work* der Rechner *Voltaire* stärker belastet wurde. Zumindest für die Strategie *Queuelength* läßt sich das nicht dadurch erklären, daß *Voltaire* auch für derart hohe Lasten noch über Leistungsreserven verfügt oder durch die Präferenz der Strategien für den zuerst gefunden Server öfter vermittelt wird. Die Betrachtung

der Abbildung 5.16 zeigt vielmehr, daß die beiden Rechner *Titanic* und *Voltaire* bei *Queuelength* in etwa gleich oft genutzt wurden.

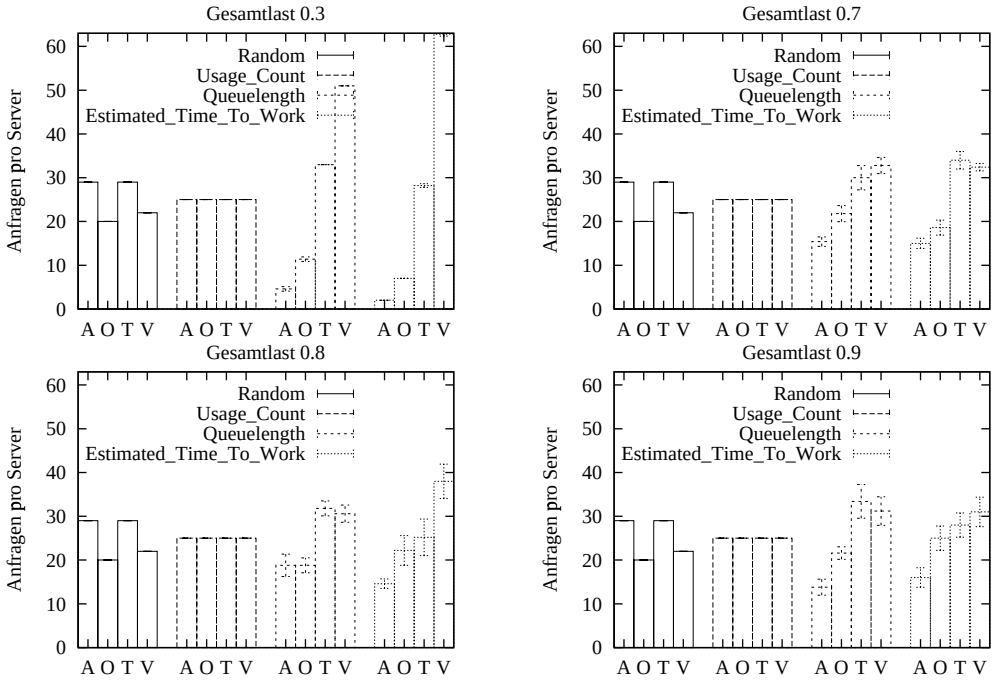

Abbildung 5.16: Anfragen pro Server (inhomogenes System mit Anfrageklassen)

Wie schon beim homogenen Fall mit verschiedenen Anfrageklassen ist festzustellen, daß die Varianz der dynamischen Verfahren bezüglich der Anfragehäufigkeit relativ hoch ist. Hier liegt ebenfalls die Vermutung nahe, daß dies auf jeweils unterschiedlich ausfallende, knappe Entscheidungen innerhalb des Load Balancers zurückzuführen ist. Speziell die Schätzung der verbleibenden Antwortzeit, die die Strategie *Estimated_Time_To_Work* anhand der mittleren Bedienrate vorzunehmen versucht, wird durch die doppelte Inhomogenität dieses Szenarios noch stärker durcheinander gebracht.
Dies zeigt sich auch in Abbildung 5.17, wo für hohe Lasten die Konfidenzintervalle der *Estimated_Time_To_Work*-Strategie groß ausfallen. Das gleichermaßen schlechte Abschneiden der statischen Strategien läßt auch sich an der Anzahl der Aufträge in den jeweiligen Serversystemen festmachen. An den langsamen Rechnern entstehen große Staus, die bei der *Usage_Count*-Strategie aufgrund der dort verwendeten Gleichverteilung der Aufträge, die Inhomogenität des Szenarios wiedergeben. Die dynamischen Verfahren erreichen hingegen eine vergleichsweise gute Gleichverteilung bezüglich der einzelnen Warteschlangen.
Auch in diesem Umfeld bestätigt sich die Erkenntnis, daß bei inhomogener Rechnerleistung lediglich dynamische Strategien zur Lastverteilung geeignet sind. Durch die zusätzliche Inhomogenität, die sich durch die verschiedenen Anfrageklassen ergibt, sinkt die Leistung der *Usage_Count*-Strategie, die beim homogenen Fall mit Klassen noch relativ gut abschnitt, rapide. Als bestes Lastverteilungsverfahren erweist sich

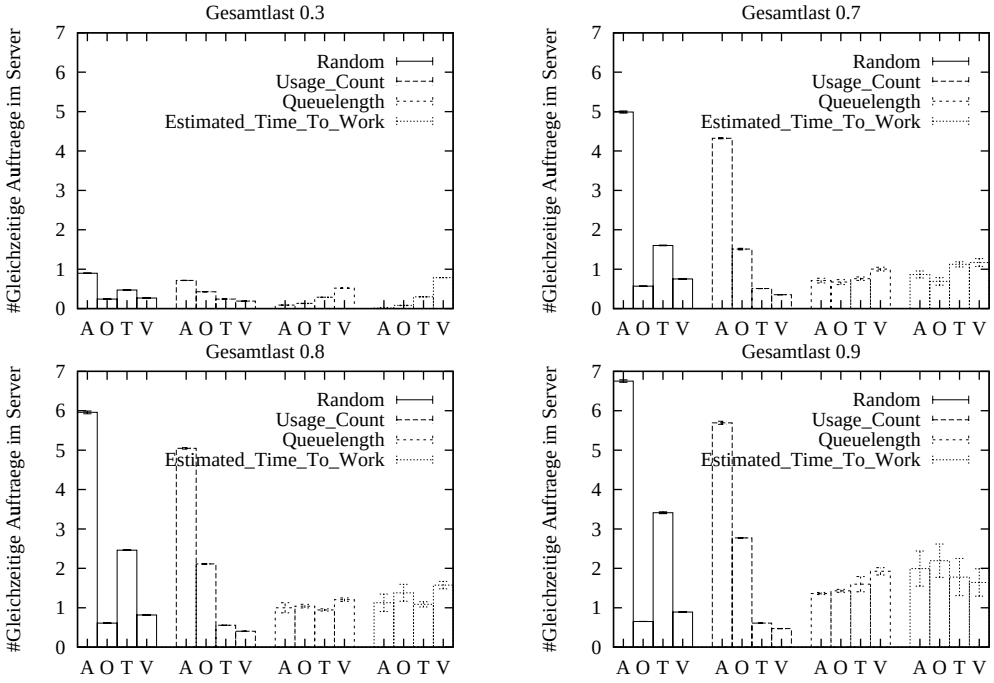

Abbildung 5.17: Aufträge im System pro Server (inhomogenes System mit Anfrageklassen)

Queuelength, die auch in diesem Szenario bei hoher Auslastung eine gleichmäßige Verteilung der entstehenden Last vornehmen kann.

5.3.2 Einfluß des Regelwerks auf die Lastbalancierung

Nachdem in den vorhergehenden Szenarien die Optimierung der Dienstauswahl lediglich bezüglich der Last stattfand, wird nun die Leistungsfähigkeit des realisierten Trader/Load Balancer-System für den Fall untersucht, daß zusätzlich bei der Optimierung die Dienstattributierung zu berücksichtigen ist. Im wesentlichen ist hierbei der Einfluß der Gewichtungsparameter und der verwendeten Norm im Regelwerk von Interesse.
Die Entscheidung, wie Last und Dienstgüte zu gewichten sind, ist subjektiv. Die folgenden Meßergebnisse können daher nur als Hilfe dienen, um zu entscheiden, welche Gewichtung gewählt werden soll, um bei den zu erwartenden Lasten noch eine akzeptable Antwortzeit zu erreichen.
Als Beispielszenario wurde der klassenlose Fall mit homogener Rechnerleistung gewählt. Die vier Dienstanbieter boten den gleichen Dienst an und wurden mit einem Dienstattribut, dessen Wert je nach Dienstanbierter unterschiedlich ausfiel, versehen. Dienstanbieter 1 bekam den Wert 25, Dienstanbieter 2 den Wert 50, Dienstanbieter 3 den Wert 75 und Dienstanbieter 4 den Wert 100.
Durch die Import-Anfrage wurde der von diesen Dienstanbietern angebotene Dienst gesucht, wobei das verwendete Dienstattribut zu minimieren war. Dem Regelwerk

wurden verschiedene Parameter für die Gewichtung von Last und Dienstattributgüte sowie für die zu verwendende Norm vorgegeben.
Da das Regelwerk neben einer Bewertung der Dienstattributgüte auch eine Bewertung der Last benötigt, können nur Lastverteilungsstrategien verwendet werden, die auf einer Lastmetrik basieren. Die Strategie *Random* scheidet daher aus. Da die Strategie *Usage_Count* die Anzahl der Dienstvermittlungen als Lastmetrik verwendet, wurde diese mit in die Betrachtung einbezogen.
Die Abbildungen 5.18 bis 5.20 und 5.22 zeigen die Antwortzeiten, die sich für die verschiedenen Lastverteilungsstrategien bei verschiedenen Gewichtungen und Normen ergeben. Neben den drei hierbei eingesetzten Strategien ist zusätzlich noch zum Vergleich das beste Ergebnis des homogenen Szenarios aus Abschnitt 5.3.1.1 eingezeichnet. Dort wurde das gleiche Umfeld verwendet, allerdings bei reiner Lastbalancierung. Die Strategie *Usage_Count* hat dort das beste Ergebnis erzielt. Es stellt für die Praxis eine realistischere Untergrenze als die des M/M/4-Warteschlangenmodells dar. An dieser muß sich das Regelwerk messen lassen.

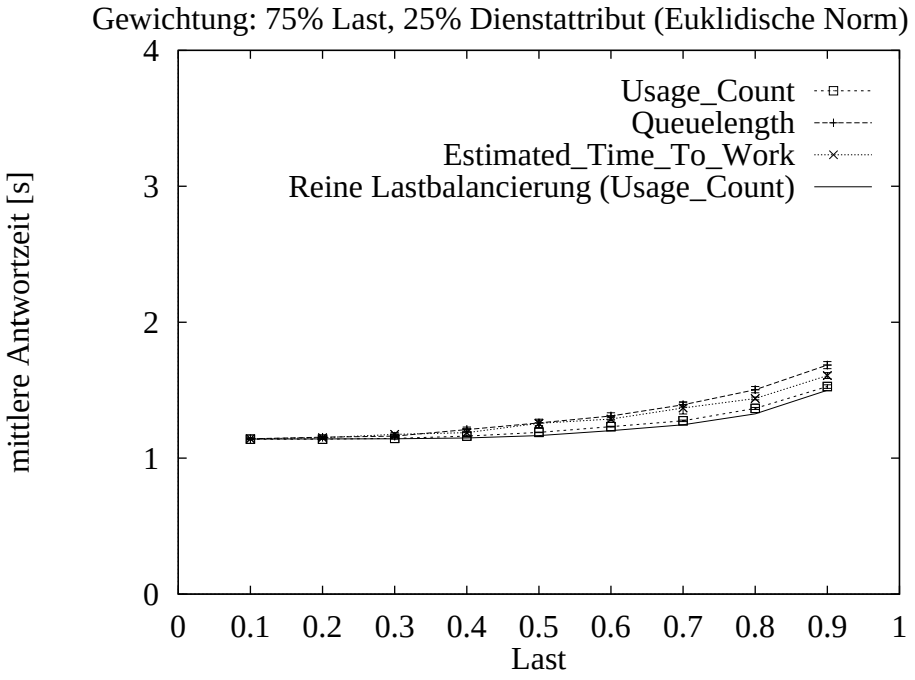

Abbildung 5.18: Antwortzeiten bei Gewichtung von 75%:25% für Last/Dienstattributgüte mit euklidischer Norm (homogenes System)

Die in Abbildung 5.18 dargestellten Antwortzeiten bei einer Gewichtung von 75% für den Lastaspekt und 25% für die Dienstattributgüte unterscheiden sich fast nicht von den Werten, die sie im Fall der reinen Lastbalancierung angenommen haben. Dort ergab sich auch schon die leicht unterschiedliche Qualität der Stragien *Usage_Count*, *Queuelength* und *Estimated_Time_To_Work*, die sich hier ebenfalls wiederfindet. Diese Gewichtung erweist sich demzufolge als unproblematisch, wenn eine Gewichtung ge-

sucht wird, bei der die Dienstgüte einfließt und weiterhin eine sehr gute Lastverteilung stattfindet.
Die abgebildeten Antwortzeiten wurden unter Verwendung der euklidischen Norm gemessen. Die Werte, die sich bei Einsatz der Manhattan-Distanz ergeben, wurden hier nicht aufgeführt, da sie sich bei der noch recht starken Gewichtung der Last nur minimal von denen der euklidischen Distanz unterscheiden. Ebensowenig wurde die Ergebnisse für eine 90% Gewichtung der Last vorgestellt, da sich das Ergebnisse der Gewichtung von 75% Last bzw. 25% Dienstgüte demgegenüber nicht verändert hat.
Bei einer 50%/50% Gewichtung zeigt sich hingegen ein anderes Bild. Sowohl die jetzt noch erzielbare Lastbalancierung, als auch die unterschiedlichen Normen betreffend, ergeben sich sichtbare Unterschiede.

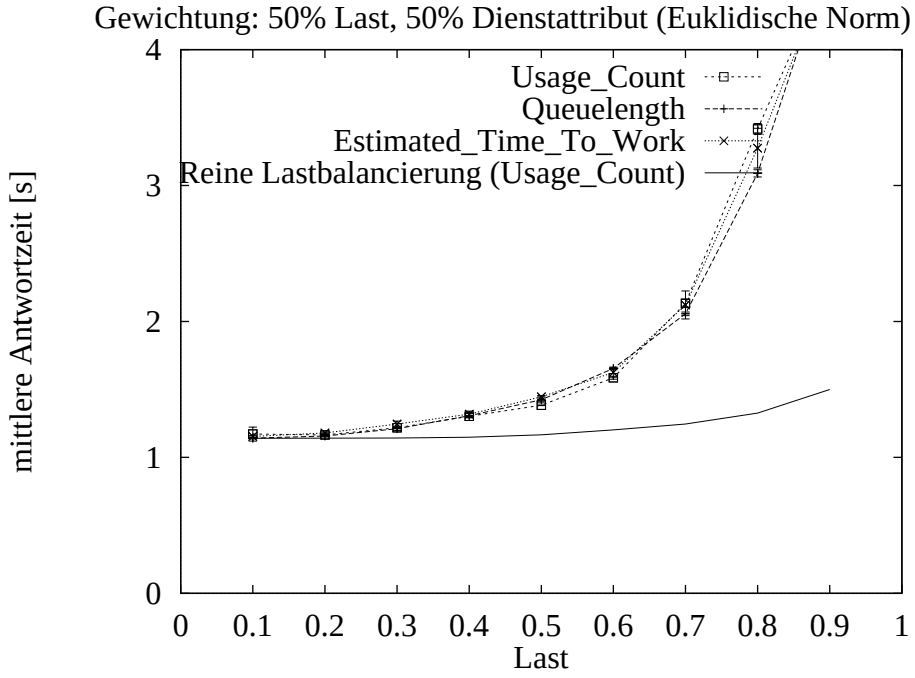

Abbildung 5.19: Antwortzeiten bei Gewichtung von 50%:50% für Last/Dienstattributgüte mit euklidischer Norm (homogenes System)

Abbildung 5.19 zeigt die Antwortzeiten bei Verwendung der euklidischen Norm. Der Anstieg der Antwortzeit erfolgt schneller als in den vorherigen Fällen. In der oberen Hälfte des Laststpektrums ergibt sich ein erheblicher Abstand von der Lastverteilungsqualität, die sich ergibt, wenn lediglich die Last berücksichtigt wird. Alle drei Strategien verhalten sich hierbei im wesentlichen gleich. Bei kleinen Lasten ergibt sich lediglich ein minimaler Vorteil für die Strategie *Usage_Count*, bei hohen für *Queuelength*.
In Abbildung 5.20 ist die analoge Situation bei Einsatz der Manhattan-Distanz aufgeführt. Die Lastverteilung erfolgt auch bei hoher Last noch gut. Dies heißt im Gegenzug aber auch, daß die Dienstattributgüte der zurückgelieferten Dienstangebote

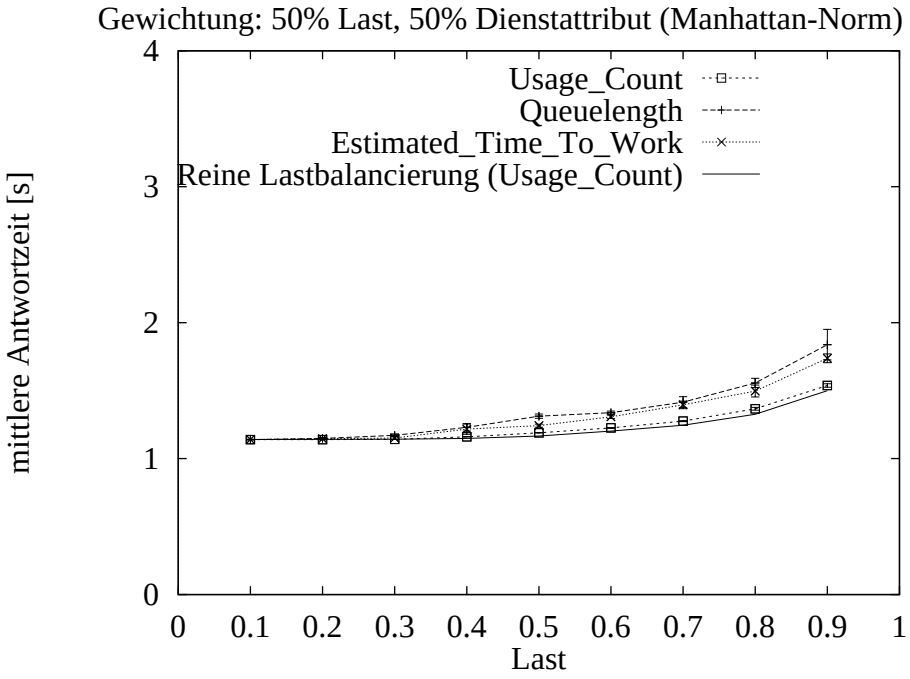

Abbildung 5.20: Antwortzeiten bei Gewichtung von 50%:50% für Last/Dienstattributgüte mit Manhattan-Norm (homogenes System)

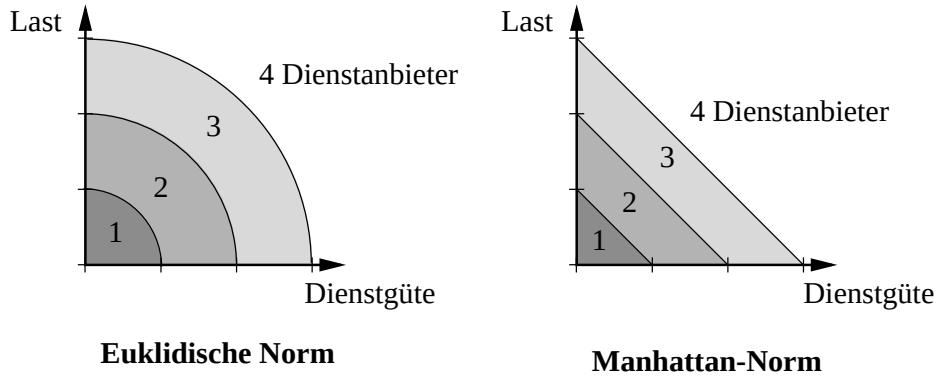

Abbildung 5.21: Anzahl der Dienstanbieter in Abhängigkeit von Lastmetrikwert und Dienstgüte

schlechter ausgefallen ist, da das Regelwerk die gegebenene Situation bzw. die Attributwerte nicht ändern kann.

Um das unterschiedliche Verhalten der beiden Normen zu verstehen, bietet es sich an, die Übergänge zu betrachten, an denen die Normen jeweils dazu übergehen, einen weiteren Dienstanbieter zu nutzen. In Abbildung 5.21 sind isometrische Linien eingezeichnet, an denen die jeweilige Norm einen weiteren Dienstanbieter mit in die Kompromißmenge aufnimmt.

Bei der euklidischen Norm ist die Fläche, innerhalb der sich ein Kompromiß zwischen Dienstgüte und Last noch auf einen Dienstanbieter beschränkt, größer als bei der Manhattan-Norm. Daher fällt bei der euklidischen Norm die Antwortzeit entsprechend höher aus. Die Manhattan-Norm geht früher dazu über, einen weiteren – von der Dienstgüte schlechteren, aber unbelasteten – Dienstanbieter vorzuschlagen. Aufgrund ihrer Charakteristik bemüht sich die Manhattan-Metrik nicht, einen Kompromiß in der Mitte zwischen Dienstgüte und Last zu finden, sondern nimmt schnell eine schlechte Diensteigenschaft in Kauf, um die Last zu senken.

Bei einer zu starken Gewichtung der Dienstattributgüte nützt dies allerdings auch nichts mehr. Abbildung 5.22 zeigt dies deutlich für den Fall, daß der Lastaspekt lediglich mit 25% gewichtet wird, die Attributgüte hingegen mit 75%. Hierbei halten beide Normen so stark an dem Dienstanbieter mit der besten Dienstattributsgüte fest, daß das Gesamtsystem zu einem System mit nur 1 Dienstanbieter degeneriert. Eine Ankunftsrate, die für ein 4-Server-System eine Last von ρ=0.3 darstellt, läßt die Antwortzeit des derart degenerierten Systems in die Höhe schnellen. Dies ist examplarisch für die euklidische Norm dargestellt.

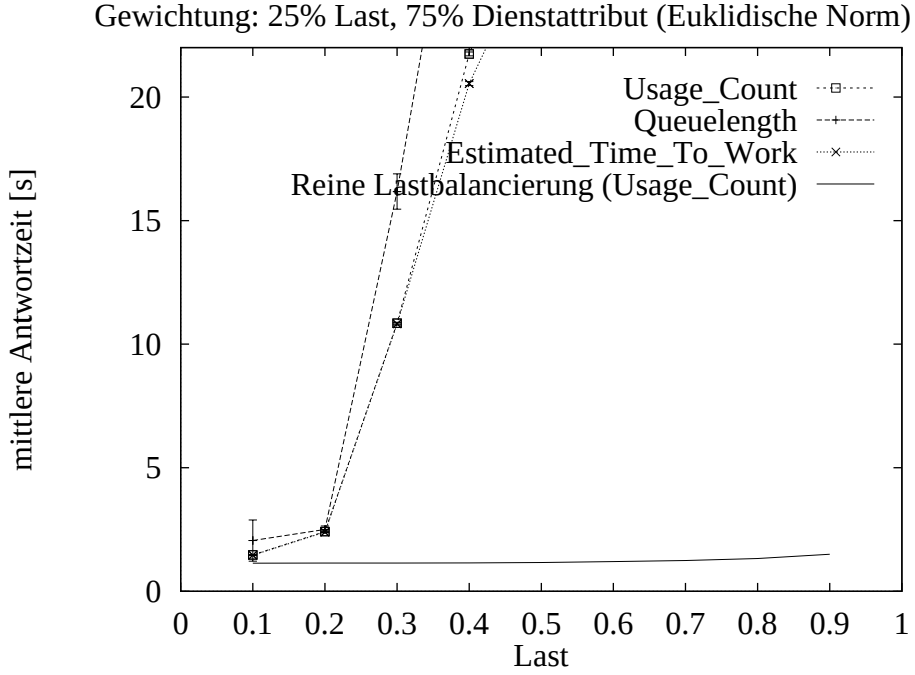

Abbildung 5.22: Antwortzeiten bei Gewichtung von 50%:50% für Last/Dienstattributgüte mit euklidischer Norm (homogenes System)

Abschließend muß darauf hingewiesen werden, daß das untersuchte Szenario willkürlich gewählt wurde. In Abhängigkeit von den Attributwerten und der Leistungsfähigkeit der Server kann sich bezüglich der Gewichtung, die noch zu akzeptablen Antwortzeiten führt, ein anderes Bild ergeben.

5.3.3 Lastverteilungsaufwand

Neben der Qualität der erzielten Lastverteilung muß zur Bewertung des realisierten Ansatzes der zusätzliche Aufwand, der sich bei der Dienstvermittlung aufgrund des Load Balancers ergibt, untersucht werden. Im folgenden werden hierzu die Dienstvermittlungszeit und die zusätzlichen Kommunikationsaufrufe, die für die Aktualisierung der Lastinformationen benötigt werden, betrachtet.

5.3.3.1 Dienstvermittlungszeit

Die Dienstvermittlungszeit setzt sich aus einem Anteil, den der Trader benötigt, um Dienstangebote aus seinen Tabellen zu suchen, und einem Anteil, den der Load Balancer benötigt, um die dabei entstehende Dienstauswahl unter dem Lastaspekt zu optimieren, zusammen.

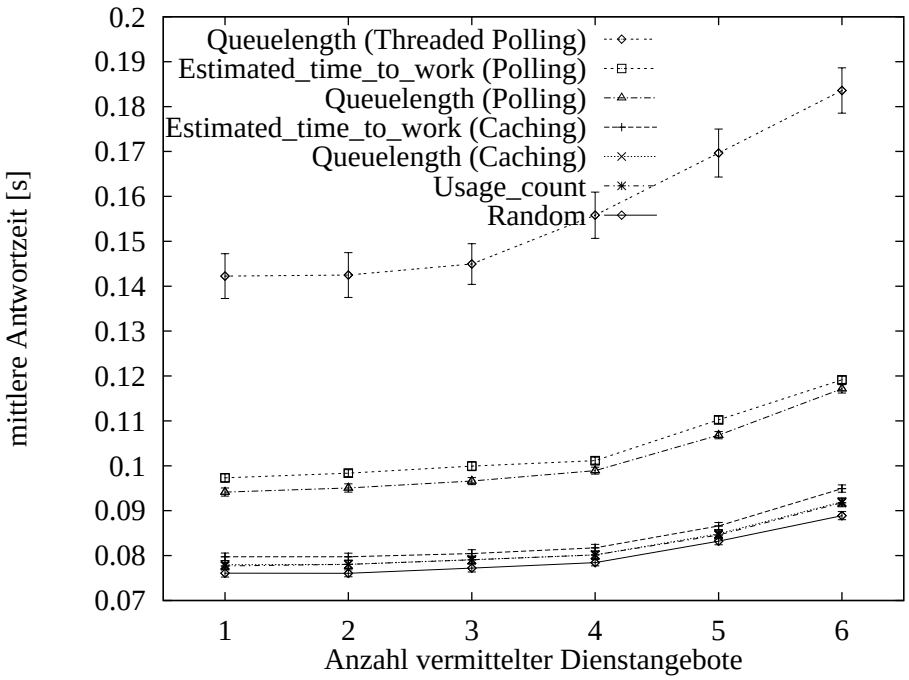

Abbildung 5.23: Mittlere Antwortzeiten der verschiedenen Trader-/Load Balancer-Varianten in Abhängigkeit von der Anzahl der zurückgelieferten Dienste

In Abbildung 5.23 ist die mittlere Antwortzeit für die Dienstvermittlung mit verschiedenen Lastverteilungsstrategien zu sehen. In der Diensttabelle des Traders waren sechs Dienstangebote eingetragen. Wie im vorhergehenden Abschnitt wurden sie über genau eine Diensteigenschaft attributiert. Jedem Dienstanbieter wurde ein eigener Diensttyp zugewiesen. Durch eine entsprechende Obertyp-/Untertyp-Relation im Typmanager war dafür gesorgt, daß bei den Import-Anfragen – je nach angegebenem Dienstobertyp – ein bis sechs Dienstangebote zurückgeliefert wurden. Sofern es von der jeweili-

gen Lastverteilungsstrategie unterstützt wurde, sollte über eine euklidische Norm ein Kompromiß zwischen Last und Dienstgüte getroffen werden.
Da die *Random*-Strategie die Dienstauswahl ohne den Load Balancer vornimmt, kann dieser Fall als Referenz für die Dienstvermittlungszeit einer reinen Traderimplementierung angesehen werden. Die Dienstvermittlungsdauer steigt in allen Varianten stärker als linear mit der Anzahl der vermittelten Dienstangebote an. Als Erklärung kommt hierfür in Frage, daß diverse Trader-interne Listen mehrfach durchlaufen werden, so daß sich insgesamt ein Aufwand $\Omega(n^2)$ ergibt.
Der Einsatz des Load Balancers und des Regelwerks führt gegenüber dem reinen Trading zu einer höheren Dienstvermittlungsdauer. Falls die benötigten Lastinformationen per Caching aktualisiert werden, fällt diese Erhöhung allerdings sehr gering aus. Die *Usage_Count* und die *Queuelength*-Strategien schneiden hierbei am besten ab. Dies ist dadurch zu erklären, daß die hierbei zugrundeliegenden Metriken sehr einfach sind. Die Strategie *Estimated_Time_To_Work* setzt eine komplexere Lastmetrik ein. Bei jedem Auslesen dieser Metrik muß eine Betriebssystemfunktion aufgerufen werden, um das Altern dieser Metrik zu berücksichtigen. Demenstprechend fällt hier die Antwortzeit etwas höher aus.
Für den Fall, daß die benötigten Lastinformationen per Polling jeweils während der Dienstvermittlung erfragt werden müssen, fällt die Erhöhung der Antwortzeit stärker aus. Zur Ermittlung der dynamischen Lastinformationen muß ein entfernter Operationsaufruf an den Monitor abgeschickt werden; dieser muß in seiner MIB die gewünschte Metrik nachschauen und zurückschicken. Die Load Balancer-Komponente muß nun ihrerseits die empfangene Last in die globale MIB eintragen. Erst dann werden die gleichen Schritte wie beim Caching-Fall durchlaufen. Die *Estimated_Time_To_Work*-Strategie nimmt noch mehr Zeit in Anspruch, da der Monitor ebenfalls das Altern dieser Metrik berücksichtigen muß.
Eine Polling-Variante, die pro Polling-Aufruf einen eigenen Thread benutzt, sollte die für das Polling benötigten Schritte parallel abarbeiten und die Pausen, die sich beim Warten auf die Antwort eines entfernten Methodenaufrufs ergeben, besser ausnutzen. Die Messung hat jedoch ergeben, daß dies nicht gelungen ist. Vielmehr scheint die Erzeugung von neuen Threads und die nötige Thread-Synchronisation so viel Zeit in Anspruch zu nehmen, daß diese Variante zu einer Verlängerung der Dienstvermittlungszeit führt. Da in einem lokalen Netz Polling-Anfragen sehr schnell beantwortet werden, ist dort eine einfache Polling-Strategie völlig ausreichend. Beim Einsatz des Load Balancers in Weitverkehrsnetzen könnte sich der Einsatz einer Polling-Strategie mit Threads hingegen lohnen, da dort die Polling-Anfragen nicht mehr so schnell beantwortet werden. Um den Overhead, der durch die Erzeugung von Threads entsteht, zu verringern, wäre es außerdem möglich, immer einen Pool von Threads bereit zu halten, auf den dann die Polling-Anfragen verteilt werden können.
Bei genauer Betrachtung der Meßergebnisse zeigt sich, daß die Dienstvermittlungszeit bei Einsatz des Load Balancers nicht um einen konstanten Betrag über der Zeit des reinen Traders mit *Random*-Strategie liegt, sondern mit steigender Anzahl der vom Load Balancer zu berücksichtigenden Dienstangebote wächst. Der Load Balancer muß je-

des vom Trader gelieferte Dienstangebot verwalten und bewerten, so daß hierfür der Aufwand $\Omega(n)$ anzusetzen ist. Da bei der Implementierung auf Effizienz geachtet wurde, geht der Load Balancer vermutlich mit $O(n)$ in die Dienstvermittlungsdauer ein. Durch die Hashfunktion wird das wiederholte Durchlaufen von Listen vermieden. Da das Regelwerk die vom Trader gelieferte Dienstangebotsmenge nicht sortieren muß, sondern lediglich den besten Kompromiß heraussucht, geht das Regelwerk nur mit dem Aufwand $O(n)$ ein.
Es läßt sich festhalten, daß die zusätzliche Lastbalancierung zu einer Verlängerung der Dienstvermittlungsdauer führt. Bei Einsatz von Caching-Verfahren zur Lastübermittlung fällt diese Verlängerung aber sehr gering aus. Polling-Strategien zur der Übertragung der Lastinformationen sollten hingegen vermieden werden.

5.3.3.2 Kommunikationsaufwand

Als letztes soll die zusätzliche Kommunikation, die für die dynamische Lastverteilung benötigt wird, betrachtet werden. Hierzu wurden die entfernten Operationsaufrufe, die zur Übermittlung der Lastinformationen verschickt werden mußten, gezählt. Beim Polling werden diese vom Load Balancer an die Monitore gesendet, beim Caching von den Monitoren an den Load Balancer.
Wie im vorherigen Abschnitt wurden sechs Dienstangebote, die in einer Untertyp-Beziehung zueinander standen, an den Trader exportiert. Durch entsprechende Import-Anfragen wurden dem Load Balancer durch den Trader – in Abhängigkeit vom gewünschten Dienstobertyp – ein bis sechs Dienstangebote vorgelegt. Zur Lastverteilung kam die *Estimated_Time_To_Work*-Strategie zum Einsatz. Diese sendet im Caching-Betrieb pro Dienstnutzung zwei Cache-Aktualisierungen an den Load Balancer. Insgesamt erfolgten einhundert Import-Anfragen mit anschließender Nutzung des vermittelten Dienstes. Eine parallele Nutzung der Dienste ohne vorherige Vermittlung durch den Trader fand nicht statt.

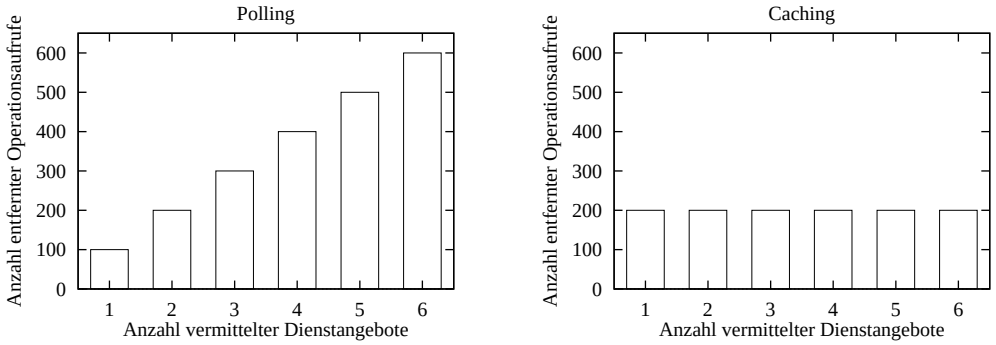

Abbildung 5.24: Anzahl der entfernten Operationsaufrufe bei der Lastübermittlung per Caching und Polling (100 Import-Anfragen und 100 Dienstnutzungen)

Die aus diesem Szenario resultierenden Ergebnisse sind in Abbildung 5.24 aufgeführt. Es zeigt sich, daß mit steigender Anzahl der Dienstanbieter, die vom Load Balancer

betrachtet werden, auch die Anzahl der Polling-Anfragen an die Monitore steigt. Bei jedem der einhundert Import-Aufrufe muß für alle in Frage kommenden Dienstanbieter die Last ermittelt werden. Die Anzahl der Caching-Nachrichten ist hingegen konstant. Ihr Wert ergibt sich aus der Anzahl der Dienstnutzungen und der Anzahl der Cache-Aktualisierungen, die pro Dienstnutzung durchgeführt werden. Die Ergebnisse entsprechen den Überlegungen, die bereits in Abschnitt 4.2.2.2 geführt wurden.
Es zeigt sich, daß bei der Übermittlung der Lastinformationen die Caching-Strategie weniger Aufrufe verursacht. Lediglich, wenn nur ein Dienstanbieter bei der Lastverteilung in Frage kommt, schneidet die Polling-Strategie besser ab. Falls eine Lastverteilungsstrategie zum Einsatz käme, die mit einer Cache-Aktualisierung pro Dienstnutzung auskommt, ergäbe sich diesen Fall ein Gleichstand.
Da eine Lastverteilung erst bei mindestens zwei Dienstanbietern möglich ist, kann – falls feststeht, daß nur ein Dienstanbieter in Frage kommt – für diesen Dienstanbieter ganz auf die Übertragung der Lastinformationen verzichtet werden. In der Praxis kann diese Entscheidung jedoch nicht allgemeingültig getroffen werden, da sich die Menge der für die Lastverteilung in Frage kommenden Dienstanbieter in Abhängigkeit vom gewünschten Dienstobertyp von Import-Anfrage zu Import-Anfrage ändern kann. Lediglich eine Betrachtung aller Diensttyp-Relationen könnte in einigen Fällen ergeben, daß es einen Dienstanbieter gibt, der unabhängig vom Obertyp seinen Dienst als einziger anbietet.
Diese und aller vorherigen Überlegungen verlieren allerdings ihre Gültigkeit, sobald ein Szenario vorliegt, in dem Cache-Aktualisierungen auch ohne vorherige Dienstvermittlung stattfinden. In diesem Fall sollte daher zur Minimierung der entfernten Operationsaufrufe der in Abschnitt 4.2.2.2 vorgestellte Automatismus zur dynamischen Umschaltung zwischen Caching und Polling eingesetzt werden.

5.4 Zusammenfassung der Meßergebnisse

Die Bewertung der Lastverteilungsgüte der verschiedenen Lastverteilungsstrategien in Abschnitt 5.3.1 hat ergeben, daß die Verwendung von Wissen des Traders über zurückliegende Dienstvermittlungen nur möglich ist, wenn ein Szenario mit homogener Serverleistung und keiner Hintergrundlast vorliegt. Davon kann in einem offenen Dienstmarkt nicht ausgegangen werden. Hierfür werden dynamische Lastverteilungsstrategien benötigt. Es konnte nachgewiesen werden, daß eine dynamische Lastverteilung, die lediglich die Warteschlangenlänge betrachtet, in verschiedensten Szenarien für eine sehr gute Lastverteilung sorgt. Weitergehendes Wissen über die Bedienrate eines Dienstanbieters hat zu keiner signifikant besseren Lastverteilung geführt.
Die Wahl von Gewichtungsfaktoren für die Dienstattributgüte und die Lastbewertung konnte nur exemplarisch betrachtet werden, da die daraus resultierende Lastverteilung von den jeweiligen Systemparametern abhängt. Es hat sich gezeigt, daß eine zu starke Berücksichtigung der Qualität der Diensteigenschaften leicht dazu führen kann, daß keine Lastverteilung mehr stattfindet. Der Vergleich zwischen euklidischer Norm

und Manhattan-Norm hat ergeben, daß die Manhattan-Norm schneller als die euklidische Norm dazu übergeht, weitere Dienstanbieter in den zu treffenden Kompromiß zwischen Last und Dienstgüte einzubeziehen. Hierbei werden schlechtere Diensteigenschaften in Kauf genommen. Der Nutzer muß entscheiden, ob dies erwünscht ist.
Bei der Betrachtung des zusätzlichen Aufwands, der durch die Berücksichtigung des Lastaspekts bei der Dienstvermittlung entsteht, hat sich ergeben, daß die Verlängerung der Dienstvermittlungsdauer gering ist, wenn zur Ermittlung der Lastinformationen eine Caching-Strategie eingesetzt wird. Polling-Strategien sollten vermieden werden, da sie neben einer erhöhten Dienstvermittlungsdauer im untersuchten Szenario auch ein erhöhtes Nachrichtenaufkommen innerhalb des Netzwerks nach sich zogen. Solange vor jeder Dienstnutzung der Trader befragt wird, verursachen Caching-Strategien einen geringeren Kommunikationsaufwand.

Kapitel 6

Schlußbemerkungen

Im Rahmen der vorliegenden Arbeit wurde ein Konzept entwickelt, um Lastverteilungsaspekte in die Dienstvermittlung eines Traders einzubeziehen.
Die Studie in Kapitel 3 hat gezeigt, daß keines der existierenden Systeme ein zufriedenstellendes Konzept hierfür anbietet.
Daher wurde ein bestehender Trader um eine Load Balancer-Komponente erweitert. Die Lastverteilung findet für den Importer transparent während der Dienstvermittlung statt. Zur Ermittlung der benötigten dynamischen Lastinformationen wurde ein auf das Monitoring beschränktes Managementsystem für die Komponenten des Verteilten Systems erstellt. Dieses unterstützt vielfältige Lastmetriken, so daß in der Load Balancer-Komponente verschiedenste Lastverteilungsstrategien benutzt werden können. Beispielhaft wurden zwei dynamische und zwei statische Lastverteilungsstrategien implementiert. Als Mittler zwischen der Trader- und der Load Balancer-Komponente kommt ein Regelwerk zum Einsatz, das die Ergebnismengen der beiden Komponenten zu einem Kompromiß zwischen hoher Dienstgüte und niedriger Last zusammenführt. Hierzu werden Dienstgüte und Last als Vektor aufgefaßt, so daß über eine Abstands-Norm derjenige Dienstanbieter ausgewählt werden kann, der dem theoretischen Optimum am nächsten kommt.
Dieses Konzept wurde, wie in Kapitel 4 beschrieben, auf der CORBA-Verteilungsplattform Orbix implementiert und unter Leistungsaspekten bewertet. Die Ergebnisse wurden in Kapitel 5 vorgestellt. Es konnten die folgenden Schlüsse gezogen werden:

- Die Optimierung der Dienstauswahl unter dem Aspekt der Last ist eine äußerst lohnenswerte Erweiterung eines Traders. Die Antwortzeiten bei der Nutzung eines mehrfach angebotenen Dienstes können hierdurch erheblich gegenüber der üblichen – rein diensteigenschaftsorientierten – Dienstvermittlung reduziert werden. Hierfür muß jedoch eine erhöhte Dienstvermittlungsdauer in Kauf genommen werden. Bei Verwendung einer Caching-Strategie zur Übermittlung der Lastinformationen fällt die Verlängerung der Dienstvermittlungsdauer aber gering aus.

- Die Verwendung von Trader-eigenem Wissen – wie die Anzahl der Vermittlungen eines Dienstanbieters – kann nur in einem idealisierten Szenario zur Lastverteilung genutzt werden. Im heterogenen Umfeld eines offenen Dienstmarktes sind statischen Lastverteilungsstrategien ungeeignet. Hierfür werden dynamische Lastverteilungstrategien benötigt. Einfache dynamische Strategien sind dabei ausreichend. Bereits die alleinige Betrachtung der Warteschlangenlängen der einzelnen Dienstanbieter garantiert eine sehr gute Lastverteilung in verschiedensten Szenarien. Genaueres Wissen über die Bedienrate eines Dienstanbieters führt demgegenüber zu keiner nennenswerten Steigerung der Lastverteilungsqualität.

- Ein Kompromiß zwischen Diensteigenschaftsgüte und Last ist möglich. Dieser führt allerdings bei zu niedriger Gewichtung des Lastaspekts wieder zu entsprechend schlechten Antwortzeiten. Die verschiedenen Normen, die im Regelwerk zur Bewertung der Dienstangebote eingesetzt werden können, besitzen unterschiedliche Charakteristiken. Die Manhattan-Norm geht schneller dazu über, Dienstangebote mit niedrigerer Dienstgüte auszuwählen. Die Gewichtung von Last und Dienstgüte und die Wahl der zu verwendenden Metrik sind eine subjektive Entscheidung und vom jeweiligen Umfeld abhängig.

Trotz der hervorragenden Ergebnisse, die erzielt werden konnten, ist der Aufwand der Implementierung eines solchen Systems nicht zu unterschätzen. Die Load Balancer-Komponente samt Monitoringsystem liegt mit ca. 4000 Quelltext-Zeilen in der gleichen Größenordnung wie der – recht einfache – Trader, auf dem das entwickelte System basiert. Das Monitoringsystem macht dabei allein die Hälfte des Umfangs aus. Es wäre daher wünschenswert, hierfür auf einer Managementumgebung innerhalb des Verteilten Systems aufbauen zu können. Die Umsetzung der entsprechenden CORBA Management-Standards erfolgt allerdings nur langsam, so daß darauf nicht zurückgegriffen werden konnte.
Die Idee, einen Trader und einen Load Balancer zu kombinieren, läßt noch über die vorgenommene Implementierung hinaus viel Spielraum für weitere Untersuchungen. Hierbei können verschiedenste Aspekte weiterverfolgt werden.
In einem großen Dienstmarkt müssen Föderationen von Tradern eingesetzt werden. Hierbei gestaltet sich die Lastverteilung komplexer. Es werden Regeln benötigt, anhand derer entschieden werden kann, wann weitere Trader befragt werden sollen, falls die Dienstanbieter der eigenen Domäne ausgelastet sind. Auch die Ermittlung von Lastinformationen ist in diesem Fall schwieriger, da nicht mehr unmittelbar auf alle benötigten Managementinformationen zugegegriffen werden kann.
Weiterhin sollten Mechanismen untersucht werden, mit denen Dienstnutzer, die nicht regelmäßig einen Trader zur Dienstauswahl aufsuchen, dennoch zur Neuwahl eines Dienstanbieters gezwungen werden können. Durch das Lösen von lang andauernden Bindungen können auch derartige Dienstnutzer von der Lastverteilung profitieren.
Statt des auf einer Vektor-Norm basierenden Regelwerks wäre es auch denkbar, Methoden aus der Fuzzy-Logik oder der Künstlichen Intelligenz einzusetzen, um die

Ergebnisse von Trader und Load Balancer zusammenzuführen. Ob sich die hiervon versprochene bessere Kompromißbildung lohnt, müßte angesichts der zu erwartenden längeren Dienstvermittlungszeit geprüft werden.
Schließlich wäre es interessant, zu untersuchen, inwiefern eine Lastverteilungskomponente einen lediglich im Binärformat vorliegenden Trader einkapseln kann, um mit dessen Hilfe eine Dienstvermittlung mit transparenter Lastverteilung vorzunehmen. Das im Kapitel 4 entwickelte Modell wäre hierfür einsetzbar, da es nicht zwingend eine Verzahnung von Trader und Load Balancer voraussetzt.

Literaturverzeichnis

[APRA98] P. Averkamp, A. Puder, K. Römer, K. Auel: *Urbi et ORBi*. In: iX Multiuser-Multitasking-Magazin, Heft 10/1998, Heise, S. 74-85

[BK96] N. Brown, C. Kindel: *Distributed Component Object Model Protocol – DCOM/1.0*. 1996 (http://www.microsoft.com)

[Bol89] G. Bolch: *Leistungsbewertung von Rechensystemen mittels analytischer Warteschlangenmodelle*. B. G. Teubner, Stuttgart, 1989

[BU96] H. Brunne, Th. Usländer: *Design of a Monitoring System for CORBA-based Applications*. In: Trends in Distributed Systems '96 – Industrial and Short Paper Proceedings, O. Spaniol, C. Linnhoff-Popien, B. Meyer (Hrsg.), Aachener Beiträge zur Informatik (ABI), Band 17, Verlag der Augustinus Buchhandlung, Aachen, 1996, S. 52-66

[Bur95] C. Burger: *Cooperation policies for traders*. In: Open Distributed Processing – Experiences with distributed environments, K. Raymond, L. Armstrong (Ed.), Chapman & Hall, 1995, S. 208-218

[Bur97] R. Burkhardt: *UML – Unified Modelling Language: Objektorientierte Modellierung für die Praxis*. Addison-Wesley-Longman, Bonn, 1997

[CFSD90] J. Case, M. Fedor, M. Schoffstall, C. Davin: *Simple Network Management Protocoll*. Request for Comments 1157, 1988

[CHY+97] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C.-Y. Wang, Y.-M. Wang: *DCOM and CORBA – Side by Side, Step by Step, and Layer by Layer*. Bell Labs, 1997 (http://www.bell-labs.com)

[CLR94] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest: *Introduction to Algorithms*. 11. Auflage, MIT Press, 1994

[CPRV96] V. Catania, A. Puliafito, S. Riccobene, L. Vita: *Monitoring performance in distributed systems*. In: Computer Communications, Band 19, Elsevier, 1996, S. 788-803

[Del97] T. Delicia: *Modeling of Some Plain Load Distribution Strategies for Jobs in a Multicomputer System*. Informatics and Computer Science, Vol. 97, No. 1/2, Elsevier / North-Holland, 1997, S. 35-43

[Die97] S. Dierkes: *Load Balancing with a Fuzzy-Decision Algorithm*. In: Informatics and Computer Science, Vol. 97, No. 1/2, Elsevier / North-Holland, 1997, S. 159-177

[EJ91] H. Esser, H. Th. Jongen: *Analysis für Informatiker*. Skript zur Vorlesung, 1. Auflage, Augustinus, 1991

[ELZ86] D. L. Eager, E. D. Lazowska, J. Zahorjan: *Adaptive Load Sharing in Homogenous Distributed Systems*. In: IEEE Transactions on Software Engineering, Vol. SE-12, No. 5, IEEE, 1986, S. 662-675

[FS98] M. Fowler, K. Scott: *UML konzentriert: die neue Standard-Objektmodellierungssprache anwenden.* 2. Auflage, Addison-Wesley-Longman, Bonn, 1998

[GGM93] M. A. Goodman, M. Goyal, R. A. Massoudi: *Solaris Porting Guide*, Sun, 1993

[GLL96] W. Golubski, D. Lammers, W. Lippe: *Theoretical and Empirical Results on Dynamic Load Balancing in an Object-Based Distributed Environment*. In: Proceedings of the 16th International Conference on Distributed Systems (ICDCS96), 1996, S. 537-544

[HA93] H.-G. Hegering, S. Abeck: *Integriertes Netz- und Systemmanagement*. Addison-Wesley, Bonn, 1993

[Hav98] B. R. Haverkort: *Performance of Computer Communication Systems: a model-based approach*. John Wiley & Sons, Chichester, 1998

[Hei95] M. Heineken: *Leistungsanalyse der Dienstvermittlung in einem ODP-Prototypsystem unter besonderer Berücksichtigung der Kommunikation*. Diplomarbeit, Lehrstuhl für Informatik IV, RWTH Aachen, 1995

[IDSM93] IDSM/SysMan: *IDSM/SysMan Management Architecture*. Hervé Barbot (Ed.), 1993

[Inp98] Inprise: http://www.inprise.com/visibroker

[ISO84] ISO/IEC IS 7498: *Information Processing System – Open Systems Interconnection – Basic Reference Model*. 1984

[ISO89] ISO/IEC IS 7498-4: *Information Processing System – Open Systems Interconnection – Basic Reference Model – Part 4: Management Framework*. 1989

[ISO91] ISO/IEC IS 10165-4: *Information Technology – Open Systems Interconnection – Structure of Management Information – Part 4: Guidelines for the Definition of Managed Objects*. 1991

[ISO94] ISO/IEC 13235: *Draft ODP Trading Function*. 1994

[ISO95a] ISO/IEC IS 10746-1: *ODP Reference Model Part 1: Overview*. 1995

[ISO95b] ISO/IEC IS 10746-3: *ODP Reference Model Part 3: Architecture*. 1995

[ISO96] ISO/IEC DIS 13235-1: *ODP Trading Function Part 1: Overview*. 1996

[Ion97a] Iona: *Orbix Programmer's Guide 2.3*. IONA Technologies, 1997

[Ion97b] Iona: *Orbix Programmer's Reference 2.3*. IONA Technologies, 1997

[Ion98] Iona: *OrbixTrader Programmer's Guide and Reference*. IONA Technologies, 1998

[JLS+87] J. Joyce, G. Lomow, K. Slind, B. Unger: *Monitoring Distributed Systems*. In: ACM Transactions on Computer Systems, Vol. 5, No. 2, 1987, S. 121-150

[Kau92] F.-J. Kauffels: *Netzwerk-Management: Probleme, Standards, Strategien*. Datacom, Bergheim, 1992

[KB95] E. Kovacs, C. Burger: *Projektbeschreibung: MELODY – Management Environment for Large Open Distributed Systems*. Institut für Parallele und Verteilte Höchstleistungsrechner, Universität Stuttgart, 1995

[Kel93] L. Keller: *Vom Name-Server zum Trader – ein Überblick über Trading in verteilten Systemen*. In: Praxis der Informationsverarbeitung und Kommunikation (PIK), Band 16, K.G. Saur Verlag, München, 1993, S. 122-133

[KN97] A. Keller, B. Neumair: *Using ODP as a Framework for CORBA-based Distributed Applications Management*. In: Proceedings of the IFIP/IEEE Joint International Conference on Open Distributed Processing (ICODP) and Distributed Platform (ICDP), Toronto, 1997

[Kov94] E. Kovacs: *Trading und Management verteilter Anwendungen: zentrale Aufgaben für zukünftige verteilte Systeme*. In: Neue Konzepte für die Offene Verteilte Verarbeitung, C. Popien, B. Meyer (Hrsg.), Aachener Beiträge zur Informatik (ABI), Band 7, Verlag der Augustinus Buchhandlung, Aachen, 1994, S. 57-66

[Kov96] E. Kovacs: *Advanced Trading Services Through Mobile Agents*. In: Trends in Distributed Systems '96 – Industrial and Short Paper Proceedings, O. Spaniol, C. Linnhoff-Popien, B. Meyer (Hrsg.), Aachener Beiträge zur Informatik (ABI), Band 17, Verlag der Augustinus Buchhandlung, Aachen, 1996, S. 112-113

[KRK94] Th. Koch, G. Rohde, B. Krämer: *Adaptive Load Balancing in a Distributed Environment*. In: 1st Int. Workshop on Services in Distributed and Networked Environments, IEEE Computer Society Press, Prag, 1994, S. 115–121

[Küp95] A. Küpper: *Untersuchung von dynamischen Attributierungsansätzen bei der Dienstvermittlung unter ANSAware*. Diplomarbeit, Lehrstuhl für Informatik IV, RWTH Aachen, 1995

[LR96] C. Linnhoff-Popien, P. Reichl: *Netzmanagement – Skript zur den Vorlesungen an der RWTH Aachen und der Universität GH Essen*. Aachener Beiträge zur Informatik (ABI), Band 18, Verlag der Augustinus Buchhandlung, Aachen, 1996

[Mey90] B. Meyer: *Objektorientierte Softwareentwicklung*. Hanser/Prentice-Hall International, München/London, 1990

[MP93] Z. Milosevic, M. Phillips: *Some new performance considerations in open distributed environments*. In: IEEE International Conference on Communications ICC'93, IEEE, New York, 1993, S. 961-965

[MS93] M. Mansouri-Samani, M. Sloman: *Monitoring Distributed systems*. In: IEEE Network, 7. Jg., Heft 6, November 1993, S. 20-30

[MTS89] R. Mirchandaney, D. Towsley, J. A. Stankovic: *Analysis of the Effects of Delays on Load Sharing*. In: IEEE Transactions on Computers, Vol. 38, No. 11, IEEE, 1989, S. 1513-1525

[Nag90] M. Nagl: *Softwaretechnik: methodisches Programmieren im Großen*. Springer, Berlin, 1990

[OMG95a] OMG: *The Object Management Architecture Guide*. 1995 (http://www.omg.org)

[OMG95b] OMG: *CORBAfacilities: Common Facilities Architecture*. 1995 (http://www.omg.org)

[OMG97] OMG: *CORBAservices: Common Object Service Specification*. 1997 (http://www.omg.org)

[OMG98] OMG: *The Common Object Request Broker: Architecture and Specification – Rev. 2.2*. 1998 (http://www.omg.org)

[OSF94] OSF: *OSF DCE Application Development Guide*. Cambridge, MA., 1994 (http://www.osf.org)

[Pop96] C. Popien: *Verteilte Systeme – Skript zur den Vorlesungen an der RWTH Aachen und der Universität GH Essen*. Aachener Beiträge zur Informatik (ABI), Band 16, Verlag der Augustinus Buchhandlung, Aachen, 1996

[PR98] A. Puder, K. Römer: *MICO is CORBA*. dpunkt, Heidelberg, 1998

[Ray94] K.A. Raymond: *Reference Model of Open Distributed Processing: a Tutorial*. In: Open Distributed Processing II, J. de Meer et.al., Elsevier / North-Holland, 1994, S. 3-14

[RBP+93] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Objektorientiertes Modellieren und Entwerfen*. Hanser/Prentice-Hall International, München/London, 1993

[Red96] J.-P. Redlich: *CORBA 2.0: Praktische Einführung für C++ und Java*. Addison-Wesley, Bonn, 1996

[SB97] B. Schiemann, L. Borrmann: *A new approach for load balancing in high-performance decision support systems*. In: Future Generations Computer Systems, No. 12, Elsevier / North-Holland, 1997, S. 345-355

[Sch91] A. Schill: *Verteilte objektorientierte Systeme: Grundlagen und Erweiterungen*. In: Informatik Forschung und Entwicklung, Band 6, Springer, 1991, S. 14-27

[Sch96a] B. Schiemann: *A New Approach for Load Balancing in Heterogenous Distributed Systems*. In: Trends in Distributed Systems '96 – Industrial and Short Paper Proceedings, O. Spaniol, C. Linnhoff-Popien, B. Meyer (Hrsg.), Aachener Beiträge zur Informatik (ABI), Band 17, Verlag der Augustinus Buchhandlung, Aachen, 1996, S. 29-39

[Sch96b] B. Schiemann: *Specification of IDL Mechanisms for Supporting Load Balancing*. LYDIA / WP.4 / T4.2 / D6, ESPRIT III P8144, 1996

[Sch97a] B. Schiemann: *Exploiting Interface Definition Languages for Load Balancing*. In: Informatics and Computer Science, Vol. 97, No. 1/2, Elsevier / North-Holland, 1997, S. 221-231

[Sch97b] D. Schäfer: *Entwurf und Bewertung von Caching-Strategien bei der Anfrageauswertung in Traderföderationen*. Diplomarbeit, Lehrstuhl für Informatik IV, RWTH Aachen, 1997

[Sei94] J. Seitz: *Netzwerkmanagement*. International Thomson Publishing, Bonn, 1994

[Sem97] M. Semrau: *Dynamisches Load Balancing für replizierte CORBA-Objekte*. Diplomarbeit, Lehrstuhl für Informatik IV, RWTH Aachen, 1997

[SF94] O. Spaniol, A. Faßbender: *Modellierung und Bewertung von Rechensystemen*. Skript zur Vorlesung, 2. Auflage, Aachen, 1994

[SG94] A. Silberschatz, P. B. Galvin: *Operating Systems Concepts*. Addison-Wesley, Reading, 1994

[SL94] M. Schuhmann, A. Lobinger: *Zeitspiel*. In: iX Multiuser-Multitasking-Magazin, Heft 11/1994, Heise, S. 168-173

[Slo94] M. Sloman: *Network and Distributed Systems Management*. Addison-Wesley, Reading, 1994

[SPM94] O. Spaniol, C. Popien, B. Meyer: *Dienste und Dienstvermittlung in Client/Server-Systemen*. International Thomson Publishing, Bonn, 1994

[Sta95] M. Stal: *Der Zug rollt weiter*. In: iX Multiuser-Multitasking-Magazin, Heft 5/1995, Heise, S. 160-168

[Stö93] H. Stöcker: *Taschenbuch mathematischer Formeln und moderner Verfahren*. 2. Auflage, Harri Deutsch, Frankfurt, 1993

[Str92] B. Stroustrup: *Die C++-Programmiersprache*. 2. Auflage, Addison-Wesley, Bonn, 1992

[Tan95] A. S. Tanenbaum: *Moderne Betriebssysteme*. 2. Auflage, Hanser/Prentice-Hall International Editions, München/London, 1995

[Tan96] A. S. Tanenbaum: *Computer Networks*. Prentice-Hall International, London, 1996

[Thi96] D. Thißen: *QoS-basierte Optimierung der Dienstselektion in einem ORBIX-Trader*. Diplomarbeit, Lehrstuhl für Informatik IV, RWTH Aachen, 1996

[WM85] Y.-T. Wang, R. J. T. Morris: *Load Sharing in Distributed Systems*. In: IEEE Transactions on Computers, Vol. C-34, No. 3, IEEE, 1985, S. 204-217

[WT93] A. Wolisz, V. Tschammer: *Performance aspects of trading in open distributed systems*. In: Computer Communications, Band 16, Butterworth-Heinemann, 1993, S. 277-287

[YD96] Z. Yang, K. Duddy: *CORBA: A Platform for Distributed Object Computing*. DSTC, Australien, 1996

[ZFD93] M. Zimmermann, M. Feldhoffer, O. Drobnik: *Verteilte Anwendungen: Entwurf und Realisierung*. In: Praxis der Informationsverarbeitung und Kommunikation (PIK), Band 16, K.G. Saur Verlag, München, 1993, S. 62-69

[Zla96] S. Zlatintsis: *Entwurf und Bewertung eines Trader-Gateway zwischen ANSAware und ORB-Systemen*. Diplomarbeit, Lehrstuhl für Informatik IV, RWTH Aachen, 1996