Ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen

Markus Lindermeier

Fakultät für Informatik
der Technischen Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation

Ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen

Markus Lindermeier

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. H. M. Gerndt

Prüfer der Dissertation:

1. Univ.-Prof. Dr. A. Bode

2. Univ.-Prof. (Komm. Leiter) Dr. E. Jessen, em.

Die Dissertation wurde am 25.03.2002 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29.05.2002 angenommen.

Kurzfassung

Verteilte Systeme haben in den letzten Jahren sowohl in der Industrie als auch in der Forschung stark an Bedeutung gewonnen. Im Lauf der Zeit sind unterschiedliche Programmiermodelle zur Entwicklung verteilter Anwendungen entstanden. Diese reichen von Systemen zum Nachrichtenaustausch bis hin zu verteilten objektorientierten System, die vorläufig am Ende dieser Entwicklung stehen. Eines der größten Probleme in diesem Bereich ist Lastungleichheit, die dazu führen kann, daß die Ressourcen des verteilten Rechensystems nicht vollständig genutzt werden. Dies wirkt sich negativ auf das Laufzeitverhalten verteilter Anwendungen aus, da ein Teil der Leistung des verteilten Rechensystems ungenutzt bleibt. Lastverwaltungssysteme helfen Lastunterschiede zu verringern und tragen dadurch zur Verbesserung des Laufzeitverhaltens verteilter Anwendungen bei. Ziel dieser Arbeit ist es, ein Konzept zur Lastverwaltung in verteilten objektorientierten System zu entwickeln, das den besonderen Anforderungen und Möglichkeiten dieses Programmiermodells gerecht wird.

Ausgehend von früheren Ansätzen zur Klassifikation von Lastverwaltungssystemen wird ein erweitertes Klassifikationsschema entwickelt, das den Anforderungen verteilter objektorientierter Systeme genügt. Die Klassifikation ist in die drei Teilbereiche Lasterfassung, Lastbewertung und Lastverteilung gegliedert. Mit Hilfe dieses Klassifikationsschemas wird aus den speziellen Eigenschaften verteilter objektorientierter Systeme ein Konzept zur Lastverwaltung abgeleitet. Die wesentlichen Innovationen dieses Konzepts sind die Transparenz der Lastverwaltung, die Unterstützung zustandsbehafteter Objekte und die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen. Zur Beschreibung konkreter Strategien zur Lastbewertung wird ein Lastmodell für verteilte objektorientierte Systeme vorgestellt. Dieses Modell dient als Grundlage für einen Lastbewertungsalgorithmus, der die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen unterstützt.

Um die praktische Relevanz dieses Ansatzes aufzuzeigen, wird die Common Object Request Broker Architecture (CORBA) ausgewählt. Das beschriebene Konzept ist in dem Lastverwaltungssystem LMC (Load Managed CORBA) realisiert. Wie Untersuchungen in einer Testumgebung und eine Fallstudie aus dem Bereich der medizinischen Bildverarbeitung zeigen, ist das vorgestellte Lastverwaltungskonzept geeignet, Lastunterschiede in verteilten objektorientierten System auszugleichen und dadurch das Laufzeitverhalten verteilter Anwendungen maßgeblich zu verbessern. Als besonders wirkungsvoll erweist sich dabei die Betrachtung zustandsbehafteter Objekte und die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen.

Danksagung

An dieser Stelle möchte ich allen danken, die zum Gelingen dieser Arbeit beigetragen haben. Allen voran danke ich meiner Frau Christine, die mir stets mit sehr viel Geduld und Verständnis zur Seite steht. Besonders herzlich bedanke ich mich auch bei meinen Eltern, die mir diesen Weg ermöglicht haben.

Mein besonderer Dank gilt meinen Doktorvater Prof. Dr. Arndt Bode, der allen Mitarbeitern am Lehrstuhl für Rechnertechnik und Rechnerorganisation ein sehr kreatives und innovatives Arbeitsumfeld schafft. Ich danke ihm sehr herzlich für sein Engagement und seine fachliche Betreuung. Prof. Dr. Eike Jessen danke ich für die Begutachtung meiner Arbeit. Durch seine wertvollen Hinweise gelang es, Unklarheiten zu beseitigen und Aussagen zu präzisieren.

Meine Arbeit war in das Graduiertenkolleg “Kooperation und Ressourcenmanagement in verteilten Systemen” eingebunden. In diesem Zusammenhang danke ich Prof. Dr. Wilfried Brauer für die Möglichkeit am Graduiertenkolleg teilnehmen zu können und allen Kollegen für die zahlreichen Diskussionen.

Ich möchte auch allen Kollegen vom Lehrstuhl für Rechnertechnik und Rechnerorganisation danken, die durch viele Diskussionen und Anregungen wesentlich zum Gelingen dieser Arbeit beigetragen haben. Allen voran danke ich Prof. Dr. Thomas Ludwig und dem Leiter der Tools-Gruppe Dr. Roland Wismüller, die meine Arbeit mit sehr viel Engagement begleitet und gefördert haben. Mein besonderer Dank gilt den Kollegen aus der Tools-Gruppe, Dr. Günther Rackl, Dr. Jörg Trinitis und Dr. Christian Röder. Unsere Diskussionen haben die Richtung und den Werdegang meiner Arbeit stark beeinflußt.

Allen Studenten, die in Diplomarbeiten und Systementwicklungsprojekten Teile der Implementierung durchgeführt haben, danke ich für ihr Engagement. Stellvertretend möchte ich Andreas Schmidt, Alexandros Stamatakis, Jörn Eichler, Michael Rudorfer, Michael Müller und David Laabs nennen.

Abschließend danke ich Klaus Tilk, dem Systemadministrator am Lehrstuhl für Rechnertechnik und Rechnerorganisation, für die hervorragende technische Unterstützung. Ich möchte auch den Sekretärinnen Frau Hinterwimmer, Frau Nishnik und Frau Brunnhuber für die unkomplizierte Erledigung vieler verwaltungstechnischer Angelegenheiten danken.

Inhaltsverzeichnis

Abbildungen

Tabellen

Definitionen

1. Einleitung

Die Geschichte der Informatik ist im Vergleich zu anderen Disziplinen sehr kurz. Trotzdem war die Informatik in dieser vergleichsweise kurzen Zeitspanne einem sehr starken Wandel ausgesetzt, dessen Triebfeder die sich stetig ausweitenden Anwendungsgebiete und Problemstellungen waren. Zur Lösung neuer und anspruchsvollerer Probleme wurden immer umfangreichere Software-Systeme entwickelt, die immer leistungsfähigere Hardware benötigten. Der wachsende Umfang und die zunehmende Komplexität der Anwendungen erforderten stets neue Methoden und Werkzeuge zur Software-Entwicklung. Anfangs wurden einfache Assembler-Sprachen eingesetzt, zu denen im Lauf der Zeit prozedurale und schließlich auch objektorientierte Programmiersprachen hinzugekommen sind, die vorläufig am Ende dieser Entwicklung stehen.

Eine der gravierendsten Veränderungen in der Geschichte der Informatik wurde durch die Entstehung der Netzwerktechnik und ihre starke Verbreitung eingeleitet. Anwendungen waren nicht länger auf einen einzelnen Rechner beschränkt. Die Möglichkeit zur Kommunikation über Rechnergrenzen hinaus führte zur Entstehung von verteilten Anwendungen. Diese übertreffen herkömmliche Anwendungen hinsichtlich ihrer Komplexität und ihres Umfangs bei weitem. Aus diesem Grund sind im Lauf der Zeit eine Vielzahl von Programmiermodellen zur Entwicklung verteilter Anwendungen entstanden. Diese reichen von einfachen nachrichtenorientierten Systemen über prozedurale, bis hin zu verteilten objektorientierten Systemen. Die Entwicklung der Programmiermodelle ist durchaus mit derjenigen der Programmiersprachen vergleichbar. Das Ziel der jeweiligen Programmiermodelle ist es, den Abstraktionsgrad der Kommunikation an den der zugrundeliegenden Programmiersprachen anzupassen. Ihren vorläufigen Höhepunkt findet diese Entwicklung in verteilten objektorientierten Systemen, die den Abstraktionsgrad der Kommunikation auf das Niveau der objektorientierten Programmierung heben.

Verteilte objektorientierte Systeme zeichnen sich durch einige Merkmale aus, die sie deutlich von anderen Programmiermodellen abgrenzen. Dazu zählen unter anderem das objektorientierte Programmiermodell, die Unterstützung heterogener Rechensysteme, eine offene Kommunikationsstruktur und die Transparenz der Kommunikation. Die Summe dieser Eigenschaften gewährleistet den hohen Abstraktionsgrad die-

ses Programmiermodells, der entscheidend dazu beiträgt, die Kosten für die Software-Entwicklung niedrig zu halten. Das Anwendungsspektrum verteilter objektorientierter Systeme ist äußerst breit gefächert. Es reicht von der Steuerung eingebetteter Systeme, über die Koppelung unternehmensweiter Anwendungen, bis hin zu weltweit verteilten Systemen. In den einzelnen Anwendungsbereichen kommen sehr unterschiedliche Hardware- und Software-Architekturen zum Einsatz, die bei der Betrachtung verteilter objektorientierter Systeme stets berücksichtigt werden müssen.

Ein gravierendes Problem, das bei allen verteilten Anwendungen auftritt, ist Lastungleichheit. Durch die Verteilung einer Anwendung auf unterschiedliche Rechner können Lastunterschiede im Rechensystem entstehen, so daß einzelne Rechner überlastet sind, während andere unterlastet sind, oder gar nicht genutzt werden. Dies hat zur Folge, daß ein Teil der zur Verfügung stehenden Rechenkapazität ungenutzt bleibt, was wiederum das Laufzeitverhalten der Anwendungen verschlechtert. Für die Entstehung von Lastungleichheit gibt es mehrere Ursachen:

- Häufig können verteilte Anwendungen die Rechner, auf denen sie ausgeführt werden, nicht exklusiv nutzen. Lokale Anwendungen verursachen Hintergrundlast, die zwischen den einzelnen Rechnern stark variieren kann, was letztendlich Lastunterschiede im Rechensystem nach sich zieht.

- Die Ressourcenanforderungen der Objekte einer verteilten Anwendung können sehr unterschiedlich sein. Die unterschiedliche Ressourcennachfrage führt schließlich zu einer ungleichmäßigen Auslastung des Rechensystem.

- Ein weiteres Problem ist die Heterogenität des Rechensystems. Die einzelnen Rechner unterscheiden sich häufig hinsichtlich ihrer Leistungsfähigkeit, was ebenfalls Lastunterschiede zur Folge haben kann.

- Die Arbeitslast einzelner Objekte kann stark variieren. Insbesondere kann es vorkommen, daß die Arbeitslast eines Objekts so groß ist, daß der ausführende Rechner dadurch überlastet wird. Auch dies ist eine der Ursachen für die Entstehung von Lastungleichheit.

Dem Problem der Lastungleichheit wirken Lastverwaltungssysteme entgegen. Diese versuchen eine gleichmäßige Auslastung des Rechensystems zu erreichen, indem sie die Arbeitslast zwischen über- und unterlasteten Rechnern umverteilen.

Bei der Lastverwaltung in verteilten objektorientierten Systemen müssen neben den allgemeinen Aspekten der Lastverwaltung auch die anfangs erwähnten Besonderheiten dieser Systeme betrachtet werden. Die bereits verfügbaren Lastverwaltungssysteme tun dies jedoch nur ansatzweise; vielmehr versuchen sie Lastverwaltungskonzepte anderer Programmiermodelle anzupassen und zu übertragen. Aus diesem Grund nutzen die bestehenden Lastverwaltungssysteme nur einen Teil der Möglichkeiten, die sich in verteilten objektorientierten Systemen bieten. Die Mängel der verfügbaren Lastverwaltungssysteme zeigen, daß für die Lastverwaltung in verteilten objektorientierten Systemen neue Konzepte entwickelt werden müssen, die den besonderen Anforderungen und Möglichkeiten dieser Systeme gerecht werden. Das war schließlich

die Motivation für die Entstehung dieser Arbeit, in der ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen vorgestellt wird. Wesentliche Innovationen dieses Lastverwaltungskonzepts sind die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen, die Unterstützung zustandsbehafteter Objekte und die Transparenz der Lastverwaltung. Bestehende Ansätze zur Lastverwaltung nutzen nur einen Teil der möglichen Mechanismen zur Lastverteilung. Das hier präsentierte Konzept berücksichtigt die Initialplazierung, die Migration und die Replikation. Erst durch die kombinierte Anwendung aller Lastverteilungsmechanismen wird es möglich, geeignete Maßnahmen in den vielfältigen Lastsituationen zu ergreifen, die in verteilten objektorientierten Systemen auftreten. Darüber hinaus unterstützen bestehende Lastverwaltungssysteme meist nur zustandslose Objekte. Dies schränkt das Spektrum der von der Lastverwaltung unterstützten Anwendungen erheblich ein. Letztendlich leidet darunter auch die Qualität der Lastverwaltung, da der Lastausgleich im verteilten System wesentlich schwerer zu erreichen ist. Ein weiterer Kernpunkt dieses Konzepts ist die Transparenz der Lastverwaltung. Diese gewährleistet, daß der Mehraufwand für den Entwickler so gering als möglich ausfällt, was letztendlich die Akzeptanz der Lastverwaltung erhöht. Um diese Ziele zu erreichen, wird in dieser Arbeit ein durchgehend methodischer Ansatz verfolgt, der die Thematik der Lastverwaltung ausgehend von konzeptionellen über strategische, bis hin zu technischen Aspekten beleuchtet. Schließlich wird das hier entwickelte Konzept mit anderen Ansätzen verglichen, um deren Mängel und Unzulänglichkeiten aufzudecken.

1.1 Aufbau der Arbeit

In dieser Arbeit wird schrittweise ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen entwickelt. Die Arbeit ist so strukturiert, daß sich die Problemstellung der Lastverwaltung über die einzelnen Kapitel hinweg schrittweise konkretisiert. Die Themengebiete reichen dabei von einer allgemeinen Klassifikation des Problems über den Entwurf eines geeigneten Konzepts, bis hin zu dessen strategischer Umsetzung und seiner technischen Realisierung.

Kapitel 2 beschreibt die Grundlagen verteilter Systeme und deren Programmiermodelle. Dabei werden sowohl die technischen Merkmale der unterschiedlichen Programmiermodelle als auch deren Einordnung in den historischen Kontext betrachtet. Das Spektrum dieses Kapitels reicht von nachrichtenorientierten Systemen über den entfernten Prozeduraufruf, bis hin zu verteilten objektorientierten Systemen. Der Schwerpunkt des Kapitels liegt auf den verteilten objektorientierten Systemen, ihren Objektmodellen und ihrer technischen Realisierung. Damit wird ein Begriffsgerüst geschaffen, das als Grundlage für die nachfolgenden Kapitel dient.

In Kapitel 3 wird ein Klassifikationsmodell für Lastverwaltungssysteme vorgestellt. Das Kapitel erweitert und strukturiert vorhandene Ansätze zur Klassifikation von Lastverwaltungssystemen, um daraus ein neues Klassifikationsmodell zu bilden, das auch die besonderen Anforderungen und Möglichkeiten verteilter objektorientierter Systeme berücksichtigt. Anschließend werden einige bestehende Lastverwaltungssysteme

mit Hilfe dieses Modells klassifiziert, um seine Anwendung und die daraus resultierenden Möglichkeiten zu demonstrieren.

Kapitel 4 widmet sich zuerst den besonderen Eigenschaften verteilter objektorientierter Systeme. Anhand des Klassifikationsmodells wird dann aus diesen Eigenschaften ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen hergeleitet. Dieses Lastverwaltungskonzept wird mit anderen Ansätzen zur Lastverwaltung verglichen, um die Mängel und die Schwachstellen der verfügbaren Systeme aufzuzeigen. Der zweite Teil des Kapitels beschäftigt sich mit Objekten und ihren speziellen Eigenschaften wie der Identität, der Äquivalenz und dem Zustand. Aus diesen Eigenschaften werden dann Aussagen über die Migrierbarkeit und die Replizierbarkeit abgeleitet. Diese Betrachtungen führen schließlich zum Begriff des redundanten Zustandes sowie der statischen und dynamischen Bindung.

Kapitel 5 behandelt die Lastbewertungsstrategie und führt zu diesem Zweck ein Lastmodell für verteilte objektorientierte Systeme ein. Das Lastmodell liefert eine formale Beschreibung der einzelnen Kenngrößen, die ein Lastbewertungsalgorithmus zur Entscheidungsfindung heranziehen kann. Anschließend wird mit Hilfe des Lastmodells ein Algorithmus zur Lastbalancierung entwickelt. Ziel der Lastverwaltungsstrategie ist es dabei, die Lastunterschiede bei den Ressourcen des Rechensystems möglichst gut auszugleichen und damit das Laufzeitverhalten der verteilten Anwendungen zu verbessern. Der Schluß des Kapitels befaßt sich mit der Realisierung des Lastmodells und seiner einzelnen Kenngrößen.

In Kapitel 6 wird das Lastverwaltungssystem LMC (Load Managed CORBA[1]) vorgestellt. LMC erweitert eine bestehende CORBA-Implementierung um Funktionalität zur Lastverwaltung. Dies betrifft insbesondere Problemstellungen wie die Verwaltung des Objekt-Lebenszyklus, die Objektpersistenz und die Realisierung der statischen Bindung. Alle Erweiterungen sind so konzipiert, daß sie problemlos in zukünftige Versionen des CORBA-Standards integriert werden könnten. Grundlage des Lastverwaltungssystems LMC ist sowohl das Lastverwaltungskonzept aus Kapitel 4 als auch die Lastbewertungsstrategie, die in Kapitel 5 erarbeitet wurde. Damit können die Anwendbarkeit und die Tragfähigkeit des hier präsentierten Konzepts zur Lastverwaltung in verteilten objektorientierten Systemen anhand des Lastverwaltungssystems LMC überprüft werden.

Kapitel 7 befaßt sich mit der Evaluierung des Lastverwaltungskonzepts. Zu Beginn des Kapitels werden einige allgemeine Überlegungen zur Bewertung von Lastverwaltungssystemen angestellt. Darauf folgt eine Untersuchung der Kosten des Lastverwaltungssystems LMC, um den Mehraufwand, der durch die Lastverwaltung entsteht, besser abschätzen zu können. Im Anschluß an die Kostenbetrachtung werden eine Reihe von Messungen in einer Testumgebung durchgeführt. Diese sollen die prinzipielle Arbeitsweise des Lastverwaltungssystems LMC veranschaulichen. Der Schluß des Kapitels widmet sich einer Fallstudie aus dem Bereich der medizinischen Bildverarbeitung. Diese Fallstudie demonstriert die Anwendbarkeit und den Nutzen von LMC und seinem zugrundeliegenden Lastverwaltungskonzept anhand einer praxisrelevanten Anwendung.

[1] CORBA ist die Abkürzung für Common Object Request Broker Architecture.

In Kapitel 8 wird eine kurze inhaltliche Zusammenfassung dieser Arbeit gegeben. An eine Bewertung der hier erzielten Ergebnisse schließt sich ein Ausblick auf künftige Arbeiten an.

1.2 Forschungsbeitrag der Arbeit

Die Thematik der Lastverwaltung wurde bereits für viele Programmiermodelle untersucht. Die Ergebnisse dieser Arbeiten lassen sich jedoch nicht auf verteilte objektorientierte Systeme übertragen, da sich diese sowohl in ihren Anforderungen als auch in ihren Möglichkeiten stark von anderen Programmiermodellen unterscheiden. Deshalb wird in dieser Arbeit ein neues Konzept zur Lastverwaltung entwickelt, das strikt an den speziellen Eigenschaften verteilter objektorientierter Systeme ausgerichtet ist. Dieser Abschnitt gibt einen kurzen aber dennoch vollständigen Überblick über die innovativen Aspekte dieser Arbeit und deren wissenschaftliche Relevanz.

Wegen der großen Zahl an Arbeiten im Bereich der Lastverwaltung ist eine reichhaltige aber leider auch unstrukturierte Terminologie entstanden. Dies liegt einerseits daran, daß sich viele Arbeiten auf bestimmte Anwendungen und Teilaspekte der Lastverwaltung konzentrieren und dadurch den Bezug zur übergeordneten Thematik vernachlässigen. Andererseits erfordert der stetige Wandel der Rechensysteme und ihrer Programmiermodelle eine kontinuierliche Anpassung und Erweiterung der Lastverwaltungssysteme, was eine umfassende Klassifikation historischer und aktueller Ansätze erschwert. Das Ziel des hier vorgestellten Klassifikationsmodells ist es, die Terminologie der Lastverwaltung zu vereinheitlichen und an die Anforderungen und Möglichkeiten von verteilten objektorientierten Systemen anzupassen. Zu diesem Zweck zerlegt das Klassifikationsmodell die Problemstellung der Lastverwaltung in ihre drei Teilkomponenten: Die Lasterfassung, die Lastbewertung und die Lastverteilung. Erst durch diese Aufteilung wird die Komplexität einer umfassenden Klassifikation beherrschbar.

Mit Hilfe des Klassifikationsmodells können nicht nur bestehende Lastverwaltungssysteme beschrieben werden, sondern es eignet sich auch, um neue Systeme zu planen und zu konzipieren. Dazu analysiert diese Arbeit zuerst die speziellen Eigenschaften verteilter objektorientierter Systeme, um daraus ein neues Konzept zur Lastverwaltung abzuleiten. Die wichtigsten Innovationen dieses Lastverwaltungskonzepts sind die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen, die Unterstützung zustandsbehafteter Objekte und die Transparenz der Lastverwaltung.

Bestehenden Lastverwaltungssysteme nutzen nur einen Teil der möglichen Mechanismen zur Lastverteilung. Im Gegensatz dazu verwendet das hier präsentierte Konzept sowohl die Initialplazierung, die Migration als auch die Replikation. Wie später zu sehen sein wird, ermöglicht erst die kombinierte Anwendung dieser Lastverteilungsmechanismen eine geeignete Behandlung der vielen unterschiedlichen Lastsituationen, die in verteilten objektorientierten Systemen auftreten. Eine weitere Innovation ist die Unterstützung von zustandsbehafteten Objekten. Frühere Ansätze betrachten lediglich zustandslose Objekte, was die Anwendbarkeit der Lastverwaltung stark einschränkt.

Nur durch die Betrachtung von zustandslosen und zustandsbehafteten Objekten kann der Lastverwaltung das gesamte Spektrum der verteilten objektorientierten Anwendungen zugängig gemacht werden. Ein weiterer Kernpunkt dieses Konzepts ist die Transparenz der Lastverwaltung. Transparente Lastverwaltung stellt sicher, daß der Mehraufwand für den Entwickler so gering als möglich ist, was letztendlich dazu beiträgt, die Akzeptanz der Lastverwaltung zu erhöhen.

Wie bereits erwähnt wurde, basiert dieses Lastverwaltungskonzept auf der kombinierten Anwendung der Initialplazierung, der Migration und der Replikation. Die Anwendung der Migration und der Replikation unterliegt jedoch gewissen Einschränkungen. Um diesen Sachverhalt genauer untersuchen zu können, werden zuerst die Identität, die Äquivalenz und der Zustand von Objekten betrachtet. Anhand dieser Eigenschaften und dem Ablaufprotokoll der Lastverteilung kann man dann Aussagen über die Migrierbarkeit und Replizierbarkeit von Objekten treffen. Eines der Ergebnisse dieser Betrachtungen ist der Begriff des redundanten Zustandes. Bisher wurde lediglich zwischen zustandslosen und zustandsbehafteten Objekten unterschieden. Die zusätzliche Betrachtung von Objekten mit redundantem Zustand führt zu einer wesentlichen Ausweitung der Klasse der replizierbaren Objekte, was wiederum die Einflußmöglichkeiten der Lastverwaltung erweitert. Darüber hinaus gibt es bei Objekten mit redundantem Zustand einige Optimierungen, wie die Zustandsübertragung und die statische Bindung, welche die Wirksamkeit der Replikation stark erhöhen. Insgesamt gesehen führen diese Betrachtungen zu einem erweiterten und neuen Verständnis der Migration und der Replikation, das für die kombinierte Anwendung dieser Lastverteilungsmechanismen unerläßlich ist.

Eine weitere Innovation dieser Arbeit ist ein Lastmodell für verteilte objektorientierte Systeme. Im Gegensatz zu vielen bestehenden Lastverwaltungssystemen, deren Lastbewertungsstrategie auf der Anwendung einzelner Heuristiken beruht, ermöglicht dieses Lastmodell eine ganzheitliche und formale Darstellung der Lastbewertung. Das Lastmodell ist mehrschichtig aufgebaut und berücksichtigt damit sowohl die Ebene des Rechensystems als auch die Anwendungsebene. Mit Hilfe der sogenannten Transformationsfunktion kann ein Bezug zwischen der Lastinformation der Anwendung und des Rechensystems hergestellt werden. Andere Lastmodelle verfügen über keine explizite Transformationsfunktion, was dazu führt, daß sie den Einfluß der Anwendung auf die Last des Rechensystems direkt anhand von Lastwerten ermitteln müssen. Damit wird die Lastbewertungsstrategie abhängig von der konkreten Ausprägung der Lastwerte. Das Lastmodell dieser Arbeit hingegen erlaubt die strikte Trennung der Lastbewertungsstrategie und ihrer Realisierung.

Neben diesen konzeptionellen Aspekten beinhaltet diese Arbeit auch eine Reihe von eher technisch motivierten Neuerungen. Diese betreffen insbesondere die Realisierung des Lastverwaltungskonzepts. CORBA als Zielplattform für eine Implementierung bietet nur wenig Unterstützung für die Integration der Lastverwaltung. Deshalb ist es nötig, den CORBA-Standard um einige Schnittstellen zu erweitern, die für die Realisierung des hier vorgestellten Lastverwaltungskonzepts benötigt werden. Alle Erweiterungen des CORBA-Standards fügen sich nahtlos in das zugrundeliegende Programmiermodell ein, so daß der Mehraufwand für den Entwickler sehr gering ist.

Die hier eingeführten Neuerungen betreffen insbesondere die Verwaltung von Replikatgruppen, den Objekt-Lebenszyklus, die Objektpersistenz und die Realisierung der statischen Bindung. Besonders hervorzuheben ist das Konzept der Bindungsbäume, das eine effiziente Realisierung der statischen Bindung ermöglicht, ohne dabei größere Änderungen am Kommunikationsprotokoll von CORBA vornehmen zu müssen.

2. Verteilte Systeme

Die fortschreitende Entwicklung der Rechensysteme und insbesondere deren zunehmende Vernetzung haben die technische Grundlage für die Entwicklung verteilter Systeme geschaffen. Zudem wurden durch die vermehrte Dezentralisierung und Globalisierung von Geschäftsprozessen neue Anwendungsgebiete erschlossen. Dies hat zu einer weiten Verbreitung verteilter Systeme geführt.

Die Entwicklung verteilter Systeme reicht zurück bis zum Anfang der 1980er Jahre. In diesem Zeitraum wurden verschiedene Programmiermodelle für die Entwicklung verteilter Anwendungen geschaffen. Einhergehend mit der zunehmenden Verbreitung verteilter Systeme sind auch der Umfang und die Komplexität der Anwendungen stark gestiegen. Um die Entwicklung solcher Anwendungen zu erleichtern, ist ein höherer Abstraktionsgrad des zugrundeliegenden Programmiermodells nötig. Der historische Rückblick zeigt deshalb eine kontinuierliche Steigerung des Abstraktionsgrades und eine Zunahme der Schichtungstiefe der Programmiermodelle.

Ziel dieses Kapitels ist es, einerseits einen Überblick über verteilte Systeme und die zugrundeliegenden Programmiermodelle zu geben und andererseits ein Begriffsgerüst für die nachfolgenden Abschnitte zu schaffen. Zu diesem Zweck werden zunächst die Eigenschaften und die Funktionsweise verteilter Systeme beschrieben und anschließend einige ausgewählte Programmiermodelle für verteilte Anwendungen vorgestellt. Der Fokus ist dabei sowohl auf die technischen Merkmale als auch auf die Einordnung in den historischen Kontext gerichtet. Die betrachteten Programmiermodelle reichen von einfachen Programmierbibliotheken zum Nachrichtenaustausch, bis hin zu verteilten objektorientierten Systemen mit einer Vielzahl von interagierenden Diensten.

2.1 Grundlagen

In der Literatur sind verschiedene Definitionen für verteilte Systeme zu finden [81, 39, 137, 109, 73]. Das ist auf die vielfältigen Anwendungsgebiete verteilter Systeme zurückzuführen. Für jedes Anwendungsgebiet sind bestimmte Charakteristika der Verteiltheit von besonderer Bedeutung und werden deshalb in der Definition hervorge-

hoben. Einigkeit besteht jedoch darin, daß ein verteiltes System aus einer Menge von kommunizierenden Einheiten besteht, die Berechnungen durchführen. Um der Vielfältigkeit verteilter Systeme gerecht zu werden, wird hier in Anlehnung an RÖDER [109] und MAIER [73] eine sehr allgemein gehaltene Definition verwendet:

Definition 2.1 *Verteiltes System*

Ein verteiltes System ist eine Menge von örtlich verteilten, kommunizierenden Recheneinheiten. □

Recheneinheiten können, je nachdem auf welcher Ebene man ein verteiltes System betrachtet, sowohl Hardware- als auch Software-Komponenten sein. Die unterschiedlichen Sichtweisen auf ein verteiltes System werden in Abschnitt 2.1.1 näher beschrieben. Im Hinblick auf die örtliche Verteiltheit wird zwischen eng gekoppelten und lose gekoppelten Systemen unterschieden [144]. In einem eng gekoppelten System ist die Verzögerung bei der Kommunikation niedrig und die Übertragungsrate hoch[1]. In einem lose gekoppelten System ist das Gegenteil der Fall. Diese Betrachtungsweise ermöglicht die Relativierung des Begriffs der örtlichen Verteiltheit; nicht die metrische Entfernung zweier Recheneinheiten ist entscheidend, sondern die Performanz der Kommunikation. Die Struktur und der Ablauf der Kommunikation in verteilten Systemen werden in den Abschnitten 2.1.2 und 2.1.3 beschrieben.

2.1.1 Schichtung verteilter Systeme

Verteilte Systeme bestehen sowohl aus Software-Komponenten, die Berechnungen definieren, als auch aus Hardware-Komponenten, die diese Berechnungen zur Ausführung bringen. Daraus läßt sich die Schichtung verteilter Systeme ableiten. Auf der obersten Schicht befinden sich Software-Komponenten und auf der untersten Schicht Hardware-Komponenten. Dazwischen ist eine weitere Schicht angeordnet, welche die transparente Interaktion der beiden umschließenden Schichten ermöglicht. Diese Schichtung ist in Abbildung 2.1 dargestellt [12].

Die untere Schicht beinhaltet alle Hardware-Komponenten und wird als verteiltes Rechensystem bezeichnet.

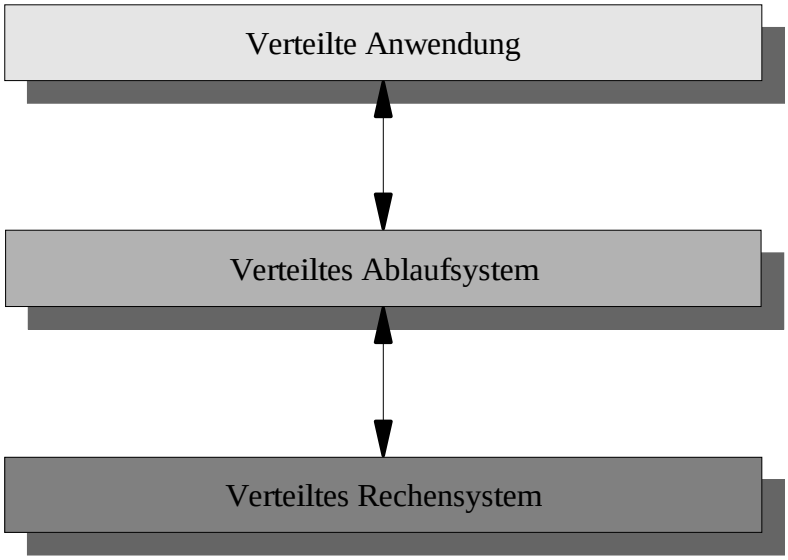
Definition 2.2 *Verteiltes Rechensystem*

Ein verteiltes Rechensystem ist eine Menge von örtlich verteilten Rechnern, die durch ein Kommunikationsnetzwerk verbunden sind. □

Als verteiltes Rechensystem können z.B. vernetzte Arbeitsplatzrechner oder Personalcomputer, aber auch Parallelrechner mit verteiltem Speicher bezeichnet werden. Die Ausdehnung des Kommunikationsnetzwerks ist beliebig; seine Performanz entscheidet darüber, ob es sich um ein eng gekoppeltes oder ein lose gekoppeltes Rechensystem handelt. Vernetzte Arbeitsplatzrechner sind in der Regel lose gekoppelt, wohingegen Parallelrechner, aufgrund ihres performanten Netzwerks, eng gekoppelt sind.

[1] Die Verzögerung der Kommunikation wird auch als Latenz und die Übertragungsrate als Bandbreite bezeichnet.

Abbildung 2.1 Die Schichten eines verteilten Systems



Die mittlere Schicht ist das verteilte Ablaufsystem. Wegen seiner Positionierung wird diese Schicht in der Literatur auch als Middleware bezeichnet.

Definition 2.3 *Verteiltes Ablaufsystem*

> Ein verteiltes Ablaufsystem ist eine Laufzeitumgebung für ein verteiltes System, die für höhere Schichten als eine virtuelle Maschine erscheint. □

Ein verteiltes Ablaufsystem ist entweder ein verteiltes Betriebssystem [38] oder es setzt auf lokalen Betriebssystemen auf und erweitert diese um Mechanismen zur Behandlung der Verteiltheit. Das Ablaufsystem schafft eine einheitliche Sicht auf das verteilte Rechensystem. Es macht sowohl die Heterogenität als auch die Verteiltheit des Rechensystems für höhere Schichten transparent. Die Verteilungseinheiten höherer Schichten werden von einer virtuellen Maschine ausgeführt ohne Kenntnis darüber zu erlangen, auf welchem konkreten Rechner dies tatsächlich geschieht. Die Kommunikation der Verteilungseinheiten läuft stets über das verteilte Ablaufsystem, da nur dieses Kenntnis über die Ausführungsorte der Kommunikationspartner hat. Das verteilte Ablaufsystem stellt Kommunikationsmechanismen zur Verfügung, die transparente Kommunikation in heterogenen Rechensystemen ermöglichen.

Der Grad an Transparenz, der bei konkreten Systemen erreicht wird, variiert sehr stark. In manchen Fällen beschränkt sich die Funktionalität des verteilten Ablaufsystems auf Mechanismen zur Kommunikation und zur Datenkonvertierung. In anderen Fällen ist die Verteiltheit des Rechensystems vollständig transparent.

Die obere Schicht eines verteilten Systems beinhaltet die Anwendungslogik und wird deshalb als verteilte Anwendung bezeichnet.

Definition 2.4 *Verteilte Anwendung*

> Eine verteilte Anwendung ist eine Menge von Verteilungseinheiten des Anwendungsbereichs, die über das verteilte Ablaufsystem kommunizieren. □

Eine verteilte Anwendung besteht aus einer Menge von Verteilungseinheiten, wie z.B. Dienste, Objekte oder Prozesse. Diese Verteilungseinheiten beinhalten die eigentliche Anwendungslogik. Das verteilte Ablaufsystem führt sie aus und ermöglicht die Kommunikation zwischen den Verteilungseinheiten.

Unabhängig von der Schichtung verteilter Systeme können auch verteilte Anwendungen in logische Schichten aufgegliedert werden. Das Modell der Gartner Group [70] identifiziert drei Schichten: Den Benutzer, die Anwendungslogik und die Datenhaltung. Je nachdem wieviele dieser Schichten auch physisch verteilt sind, spricht man von 2- oder 3-schichtigen Anwendungen. Neuere Ansätze erlauben auch eine weitere Schichtung der Geschäftslogik, was schließlich zu n-schichtigen Anwendungen führt. Die Ursache für die stetige Zunahme der Schichtungstiefe ist der wachsende Umfang und die steigende Komplexität verteilter Anwendungen. Die durch die Schichtung herbeigeführte Trennung der einzelnen Komponenten erleichtert die Entwicklung und die Wartung von Anwendungen. Die physische Verteilung ermöglicht zudem die Leistung der einzelnen Komponenten und somit die Performanz der gesamten Anwendung zu steigern.

Jedes verteilte Systeme bietet für die Entwicklung von Anwendungen entsprechende Programmierbibliotheken und Werkzeuge. Zusammen mit der Funktionalität des verteilten Ablaufsystems bilden diese ein Programmiermodell für das verteilte System. Für einen Entwickler ist der Abstraktionsgrad eines Programmiermodells ein entscheidender Faktor. Der Abstraktionsgrad bestimmt die Sichtweise des Programmierers auf das verteilte System. Einige Programmiermodelle bieten dem Entwickler lediglich Funktionalität zum Nachrichtenaustausch und verfügen somit über einen relativ geringen Abstraktionsgrad. Andere Programmiermodelle basieren beispielsweise auf objektorientierte Prinzipien und bieten dadurch einen höheren Abstraktionsgrad.

2.1.2 Organisationsformen verteilter Systeme

Ein wichtiges Merkmal verteilter Systeme ist ihre Organisationsform. Sie beschreibt die grundlegende Struktur und den Ablauf der Kommunikation zwischen den Verteilungseinheiten. Im allgemeinen wird dabei zwischen Client-Server-Systemen und Peer-to-Peer-Systemen unterschieden.

Bei Client-Server-Systemen werden die Verteilungseinheiten entsprechend ihrer Funktionalität in Clients und Server unterteilt. Server erbringen Dienste, die von den Clients angefordert werden. Ein bekanntes Client-Server-System ist das World Wide Web (WWW), das aus einer großen Menge von WWW-Servern besteht, von denen Benutzer mit Hilfe entsprechender Clients (WWW-Browser) multimediale Inhalte beziehen können.

Peer-to-Peer-Systeme zeichnen sich dadurch aus, daß sie aus gleichberechtigten Verteilungseinheiten bestehen, die sowohl Dienste anbieten als auch Dienste anderer Verteilungseinheiten in Anspruch nehmen. Die Kommunikation der Verteilungseinheiten erfolgt auf direktem Weg, ohne den Einsatz zentraler Komponenten. Peer-to-Peer Systeme haben in letzter Zeit durch die Entstehung von File-Sharing-Systemen wie Gnutella und Freenet stark an Popularität gewonnen [99].

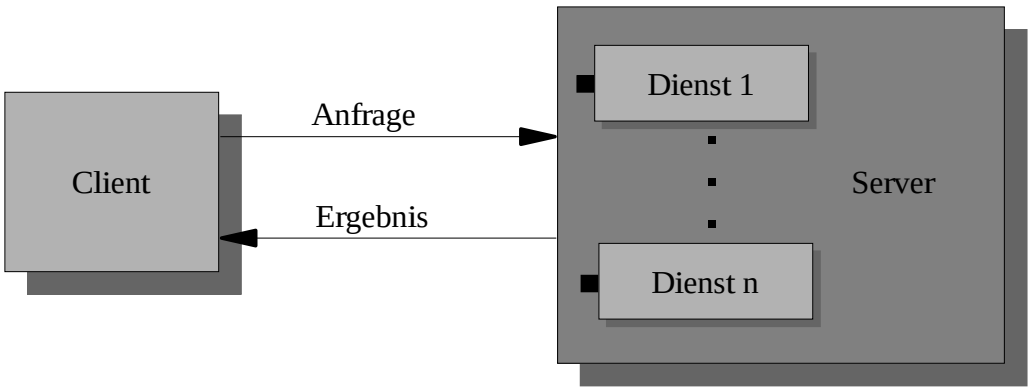
Beide Organisationsformen haben Vor- und Nachteile. Durch den Verzicht auf zentrale Komponenten ist bei Peer-to-Peer-Systemen eine gute Skalierbarkeit gewährleistet, wohingegen Client-Server-Systeme durch eine zu große Anzahl an Clients überlastet werden können. Andererseits erleichtern Client-Server-Systeme das Auffinden geeigneter Dienste, da ein zentraler Server als wohlbekannt vorausgesetzt werden kann. Letztendlich hängt die Wahl einer bestimmten Organisationsform jedoch von der Struktur und den Anforderungen der jeweiligen Anwendung ab.

2.1.3 Client-Server-Systeme

Da im Bereich der Client-Server-Systeme bereits eine wohldefinierte Terminologie vorhanden ist, wird im folgenden die Kommunikation in Client-Server-Systemen näher betrachtet [125]. Die Terminologie der folgenden Kapitel wird sich ebenfalls an Client-Server-Systemen orientieren.

Wie in Abbildung 2.2 dargestellt ist, wird ein Kommunikationspartner als Client, der andere als Server bezeichnet. Ein Server ist ein Subsystem, das eine Menge von

Abbildung 2.2 Die Kommunikation in einem Client-Server-System



Diensten zur Verfügung stellt. Meist ist ein Server als Prozeß realisiert, der auf einem Server-Rechner ausgeführt wird und als Laufzeitumgebung für seine Dienste fungiert. Die Anfragen der Clients werden vom Server den entsprechenden Diensten zugeordnet. Dienste sind die eigentlichen Ausführungseinheiten für Anfragen. Sie bieten nach außen hin eine Schnittstelle, welche die Syntax der zulässigen Anfragen beschreibt. Wenn ein Dienst eine Anfrage abgearbeitet hat, gibt er dem Client ein, der Spezifikation seiner Schnittstelle entsprechendes Ergebnis zurück. In einem verteilten System

sind Client und Server in der Regel örtlich verteilt, d.h. sie werden auf unterschiedlichen Rechnern ausgeführt.

Clients sind für Server und ihre Dienste a priori unbekannt. Das bedeutet, daß ein Server nicht feststellen kann, von welchem Client er eine Anfrage erhalten hat. Bei verbindungsorientierter Kommunikation kann ein Server zwar die Kommunikationsverbindung identifizieren aber nicht die einzelnen Clients welche diese Verbindung benutzen. Dieser Umstand muß bei der Betrachtung von Client-Server-Systemen stets berücksichtigt werden.

Kommunikation kann entweder synchron oder asynchron erfolgen [120]. Unter asynchroner Kommunikation versteht man den zeitversetzten Austausch von Nachrichten. Ein Sender verschickt eine Nachricht, die vom Empfänger zu einem beliebigen späteren Zeitpunkt entgegengenommen werden kann. Der Empfänger muß zum Zeitpunkt des Sendens nicht empfangsbereit sein. Der Sender wird nicht blockiert und kann bis zum Empfang der Nachricht weitere Berechnungen durchführen. Bei synchroner Kommunikation wird der Sender solange blockiert, bis der Empfänger die Nachricht erhalten hat. Hierbei handelt es sich also um eine zeitnahe Kommunikation. Der Sender kann bis zum Empfang der Nachricht keine weiteren Berechnungen durchführen, hat aber im Gegensatz zu asynchroner Kommunikation die Möglichkeit direkt auf Fehler zu reagieren. Bei verteilten Systemen muß berücksichtigt werden, daß die Kommunikationsmuster der Anwendung und des Ablaufsystems unterschiedlich sein können. So ist es z.B. möglich, daß asynchrone Kommunikation der Anwendung vom verteilten Ablaufsystem mittels synchroner Kommunikationsmechanismen verarbeitet wird. Bei Verwendung dieser Begriffe muß ggf. zwischen der Kommunikation auf der Ebene der Anwendung und des Ablaufsystems unterschieden werden.

Manche Client-Server-Systeme erlauben die nebenläufige Abarbeitung von Anfragen. Dabei muß unterschieden werden, ob nur verschiedene Dienste eines Servers nebenläufig aktiv sein können oder ob auch ein Dienst mehrere Anfragen gleichzeitig abarbeiten kann. Die Unterstützung von Nebenläufigkeit erhöht einerseits die Komplexität des zugrundeliegenden verteilten Ablaufsystems, da Mechanismen zur Synchronisation bereitgestellt werden müssen. Andererseits ermöglicht sie eine bessere Skalierbarkeit des Servers und seiner Dienste, indem die zur Verfügung stehenden Ressourcen besser ausgenutzt werden können.

2.2 Systeme mit Nachrichtenaustausch

Systeme mit Nachrichtenaustausch gibt es schon seit Mitte der 1980er Jahre. Die zugrundeliegenden Programmiermodelle ermöglichen es den Verteilungseinheiten durch Senden und Empfangen von Nachrichten zu kommunizieren und werden deshalb auch als Message Oriented Middleware (MOM) bezeichnet. Bei Systemen mit Nachrichtenaustausch wird zwischen Message-Queuing- und Message-Passing-Systemen unterschieden [105, 44].

2.2.1 Message-Queuing-Systeme

Die Kategorie der Message-Queuing-Systeme bietet die Möglichkeit zum Nachrichtenaustausch über Warteschlangen. Die Kommunikation erfolgt dabei verbindungslos. Der Sender legt eine Nachricht in eine Warteschlange, aus der sie der Empfänger zu einem beliebigen Zeitpunkt entnehmen kann. Dabei handelt es sich um asynchrone Kommunikation, da Senden und Empfangen einer Nachricht zeitversetzt stattfinden können.

Konkrete Implementierungen verfügen über getrennte Sende- und Empfangswarteschlangen. Das verteilte Ablaufsystem überträgt eine Nachricht von der Sendewarteschlange des Senders zur Empfangswarteschlange des Empfängers. Die Übertragung kann dabei über mehrere Zwischenstationen bzw. deren Warteschlangen erfolgen. Das verteilte Ablaufsystem garantiert die fehlerfreie Übermittlung der Nachrichten und bietet häufig auch Funktionalität zur Transaktionsverwaltung. Mechanismen zur Unterstützung heterogener Rechensysteme sind vorhanden, variieren jedoch bei konkreten Systemen sehr stark. Dem Entwickler stehen Programmierbibliotheken zur Verfügung, welche die Verwaltung lokaler Warteschlangen und die Übertragung von Nachrichten ermöglichen. Der Abstraktionsgrad des Programmiermodells ist entsprechend gering.

Aufgrund eines fehlenden Standards gibt es eine Vielzahl herstellerspezifischer Systeme. Exemplarisch seien hier MQSeries von IBM [53], das seit 1992 verfügbar ist, und MessageQ von BEA [6], das seit 1986 am Markt ist, erwähnt. Message-Queuing-Systeme kommen insbesondere im Bereich transaktionsorientierter Anwendungen mit hohen Anforderungen an die Ausfallsicherheit zum Einsatz.

2.2.2 Message-Passing-Systeme

Message-Passing-Systeme erlauben die direkte Übertragung von Nachrichten vom Sender zum Empfänger, ohne die Verwendung expliziter Warteschlangen. Die Kommunikation erfolgt verbindungsorientiert, d.h. vor der eigentlichen Kommunikation wird eine Verbindung zwischen den Kommunikationspartnern aufgebaut, die für die Dauer der Kommunikation bestehen bleibt. Somit sind die Existenz und die Empfangsbereitschaft des Empfängers garantiert.

Analog zu Message-Queuing-Systemen übernimmt das verteilte Ablaufsystem die Übertragung der Nachrichten vom Sender zum Empfänger. Für die Kommunikation in heterogenen Rechensystemen werden Mechanismen zur Datenkonvertierung angeboten. Einige Systeme bieten auch Funktionalität zur dynamischen Prozeßverwaltung und zur parallelen Ein-/Ausgabe. Dem Entwickler stehen auch hier einfache Programmierbibliotheken zur Verfügung, über die der Nachrichtenaustausch realisiert werden kann. Der Abstraktionsgrad des Programmiermodells ist, wie bei Message-Queuing-Systemen, sehr gering.

Bei Message-Passing-Systemen sind insbesondere zwei Vertreter weit verbreitet: Die Parallel Virtual Machine (PVM) [36], die seit 1991 verfügbar ist, und das Message Passing Interface (MPI) [123]. MPI ist ein Standard, der 1994 veröffentlicht wurde.

Inzwischen gibt es eine Vielzahl von Implementierungen unterschiedlicher Hersteller. Message-Passing-Systeme werden insbesondere im Bereich des Hochleistungsrechnens eingesetzt.

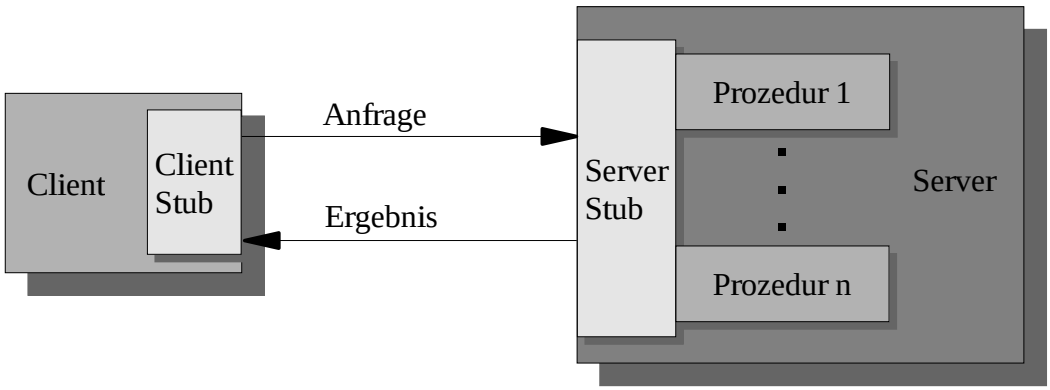
2.3 Entfernter Prozeduraufruf

Das Konzept des entfernten Prozeduraufrufs (Remote Procedure Call, RPC) reicht zurück bis in die erste Hälfte der 1980er Jahre [8]. Es ist das erste Programmiermodell, das vollständig auf Client-Server-Kommunikation ausgerichtet ist. Der RPC diente als Grundlage für viele moderne Programmiermodelle und stellt somit eine der historisch bedeutendsten Entwicklungen im Bereich der verteilten Systeme dar.

2.3.1 RPC

Der RPC überträgt das Konzept des programmiersprachlichen Prozeduraufrufs auf verteilte Systeme. Ein Server stellt Schnittstellen zur Verfügung, die von Clients über Prozeduraufrufe angefragt werden können. Dazu müssen die Schnittstellen in einer Beschreibungssprache (Interface Definition Language, IDL) definiert werden. Die IDL ist eine deklarative Sprache, deren Syntax an die Programmiersprache C angelehnt ist. Aus der Schnittstellenbeschreibung werden von einem RPC-Compiler ein Client- und ein Server-Stub erzeugt. Das sind Quelltext-Fragmente, die den Quelltexten des Clients und des Servers hinzugefügt werden. Sie ermöglichen die transparente Anbindung der Anwendung an das verteilte Ablaufsystem. In Abbildung 2.3 ist der Ablauf eines entfernten Prozeduraufrufs schematisch dargestellt. Eine Anfrage des Clients an eine entfernte Prozedur mündet im Client-Stub. Dieser überträgt die An-

Abbildung 2.3 Schematische Darstellung eines entfernten Prozeduraufrufs



frage über das verteilte Ablaufsystem zum Server-Stub. Der Server-Stub ruft schließlich die entsprechende Prozedur des Servers auf. Das Ergebnis wird auf umgekehrtem Weg an den Client zurückgegeben.

Lokale und entfernte Prozeduraufrufe sind syntaktisch identisch. Sie unterscheiden sich jedoch semantisch hinsichtlich der Behandlung von Referenzparametern. Die Werte von Referenzparametern werden bei einem entfernten Prozeduraufruf ungültig, da Client und Server unterschiedliche Adreßräume haben. Eine Referenz in den Adreßraum des Clients ist somit im Adreßraum des Servers undefiniert.

Das verteilte Ablaufsystem ermöglicht die transparente Kommunikation in einem heterogenen Rechensystem. Dazu werden die Parameter des entfernten Prozeduraufrufs im Client-Stub in ein systemunabhängiges Datenformat umgewandelt. Im Server Stub werden die Parameter wieder in das lokale Maschinenformat transformiert. Mit dem Ergebnis wird analog verfahren. Ein in diesem Zusammenhang weit verbreitetes Datenformat ist die External Data Representation (XDR) von Sun. Die Datenkonvertierung ist für den Entwickler transparent. Somit ermöglicht das Programmiermodell des RPC erstmals vollständig transparente Kommunikation in einem heterogenen Rechensystem. Zudem ist die Kommunikation syntaktisch identisch mit lokalen Prozeduraufrufen und läßt sich dadurch nahtlos in Anwendungen integrieren. Der Abstraktionsgrad des Kommunikationsmechanismus ist höher einzustufen als bei Message-Oriented-Middleware.

Erste Implementierungen von entfernten Prozeduraufrufen erschienen im Jahr 1984 [8]. Die bekannteste Implementierung ist Sun RPC, die von Sun im Jahr 1985 als Teil des Open Network Computing (ONC) Standards veröffentlicht wurde. Eine weit verbreitete Anwendung des RPC ist das Network File System (NFS) von Sun. Mittlerweile wurden RPC [132] und XDR [133] von der Internet Engineering Task Force standardisiert.

2.3.2 DCE

Das Distributed Computing Environment (DCE) [113] ist ein Programmiermodell, das auf dem Konzept des entfernten Prozeduraufrufs aufsetzt. Es definiert ein Architekturmodell für verteilte Anwendungen, das aus einer Menge von hierarchisch strukturierten Diensten besteht. Dabei können Dienste höherer Ebenen die Funktionalität der darunterliegenden Ebenen nutzen.

Auf unterster Ebene, d.h. über dem lokalen Betriebssystem und den Transportdiensten, ist der Thread Service angesiedelt. Er basiert auf dem POSIX Standard 1003.1c [84] und bietet Funktionalität zur Nutzung von leichtgewichtigen Prozessen (Threads). Auf dem Thread Service setzt der RPC auf, der als Kommunikationsmechanismus dient. Diese Kombination ermöglicht die Entwicklung von Servern, die Anfragen nebenläufig abarbeiten können. Über diesen grundlegenden Diensten sind eine Reihe höherwertiger Dienste wie z.B. der Time Service, der Directory Service, der Security Service und der Distributed File Service angeordnet. DCE Anwendungen selbst werden in Zellen unterteilt. Zellen sind Verwaltungseinheiten, in denen eine beliebige Anzahl von Benutzern, Rechnern oder Anwendungen zusammengefaßt werden. Als wesentliche Innovation gegenüber der Basistechnologie RPC bietet DCE Dienste an, die es dem Entwickler erlauben, häufig auftretende Probleme auf eine standardisierte

Art und Weise zu lösen. Aus diesem Grund ist der Abstraktionsgrad des Programmiermodells im Vergleich zu reinem RPC als wesentlich höher einzustufen.

DCE wird von der Open Group[2] standardisiert. Eine Referenzimplementierung von DCE 1.0 wurde 1992 freigegeben. In späteren Versionen wurden einige objektorientierte Erweiterungen in den Standard aufgenommen.

2.4 Verteilte objektorientierte Systeme

Verteilte objektorientierte Systeme heben den Abstraktionsgrad der Client-Server-Kommunikation auf das Niveau der objektorientierten Programmierung. Anwendungen bestehen aus einer Menge von Verteilungseinheiten, die als Server oder als Client fungieren. Server sind Objekte, die Dienste in Form von Methoden anbieten. Clients fragen diese Dienste über den Mechanismus des entfernten Methodenaufrufs an. Ein Objekt kann wiederum als Client auftreten, indem es die Dienste eines anderen Objekts nutzt. Verteilte objektorientierte Systeme basieren im allgemeinen auf einem spezifischen Objektmodell, das die Beschaffenheit der Objekte und den Ablauf der Client-Server-Kommunikation beschreibt.

Die Methoden, die ein Objekt anbietet, werden analog zum RPC mit Hilfe einer Schnittstellenbeschreibungssprache (IDL) definiert. In verteilten objektorientierten Systemen ist die Syntax der IDL meist an die Programmiersprache C++ angelehnt. Die Schnittstellen aller öffentlich zugreifbaren Methoden eines Objekts werden in IDL beschrieben. Ein IDL-Compiler erzeugt daraus Client- und Server-Stubs in der Programmiersprache der jeweiligen Zielplattform. Der Client-Stub enthält für jede Methode eines entfernten Objekts einen Stellvertreter mit identischer Schnittstelle. Dadurch wird der Client mit dem verteilten Ablaufsystem verbunden. Als Gegenstück dazu dient der Server-Stub, der das Objekt an das verteilte Ablaufsystem anbindet.

Verteilte Objekte werden, analog zu programmiersprachlichen Objekten, über Objektreferenzen adressiert.

Definition 2.5 *Objektreferenz*

> Eine Objektreferenz ist ein weltweit eindeutiger Identifikator für ein Objekt. □

Die Adressierung geschieht nach dem Broker Entwurfsmuster [14]. Ein Broker ist eine logisch zentrale Einheit des verteilten Ablaufsystems, die als Vermittler zwischen Clients und Objekten auftritt. Der Broker ist in der Lage Objektreferenzen zu erzeugen und Methodenaufrufe an Objekte bzw. deren Server zu vermitteln. Dieser Mechanismus gewährleistet die Ortstransparenz im verteilten System. Ortstransparenz ist ein wichtiges Charakteristikum verteilter objektorientierter Systeme. Sie fordert, daß einem Client der Ausführungsort eines Objekts nicht bekannt sein darf. Ein Objekt wird

[2] Die Open Group ist durch den Zusammenschluß von X/Open und der Open Software Foundation entstanden.

lediglich durch seine Objektreferenz identifiziert; die Adressierung des physischen Objekts übernimmt der Broker.

In konkreten Systemen werden Broker meist verteilt realisiert, da eine zentrale Komponente die Skalierbarkeit stark beeinträchtigen würde. Für eine verteilte Implementierung müssen Objektreferenzen sämtliche Informationen enthalten, die zur Lokalisierung eines Objekts nötig sind. Um ein spezielles Objekt zu referenzieren, erzeugt ein Client eine Instanz des Client-Stubs dieses Objekts. Diese wird mit der entsprechenden Objektreferenz parametrisiert. Dadurch entsteht im Client ein lokaler Stellvertreter des entfernten Objekts. Bei einem entfernten Methodenaufruf benutzt der Client die entsprechende Methode des lokalen Stellvertreters. Dieser lokalisiert das entfernte Objekt anhand der Objektreferenz und überträgt die Anfrage mit Hilfe des verteilten Ablaufsystems zum Server. Der Server-Stub nimmt die Anfrage entgegen, identifiziert das entsprechende Objekt und führt einen lokalen Methodenaufruf durch. Das Ergebnis wird auf umgekehrtem Weg zum Client übertragen und als Ergebnis des Aufrufs der Stellvertreter-Methode zurückgegeben. Das verteilte Ablaufsystem sorgt bei der Übermittlung entfernter Methodenaufrufe dafür, daß die Parameter und das Ergebnis entsprechend der Anforderungen des Client- und des Server-Rechners kodiert werden und verdeckt somit die Heterogenität des Rechensystems.

Der Mechanismus des entfernten Methodenaufrufs gewährleistet die Zugriffstransparenz im verteilten System. Zugriffstransparenz fordert, daß alle Objekte auf dieselbe Art und Weise angefragt werden, unabhängig von ihrer Implementierung, der verwendeten Programmiersprache und der Beschaffenheit des Rechensystems. Diese Forderung wird vom entfernten Methodenaufruf erfüllt, da sowohl die Verteiltheit als auch die Heterogenität des Rechensystems und der Anwendung verborgen bleiben. Eine weitere grundlegende Eigenschaft verteilter objektorientierter Systeme ist ihre Offenheit, die auf die Verwendung von Objektreferenzen zurückzuführen ist. Objektreferenzen werden zwar ausschließlich vom Broker erzeugt, können aber von Clients beliebig vervielfältigt und weitergereicht werden. Dies kann auch außerhalb der verteilten Anwendung geschehen, indem eine Objektreferenz beispielsweise in einer Datei gespeichert wird. Offenheit bedeutet in diesem Zusammenhang, daß es keine Einschränkungen bezüglich des Zugriffs auf Objekte gibt. Dadurch verliert das verteilte Ablaufsystem die Kontrolle über die im Umlauf befindlichen Objektreferenzen, was wiederum das Auffinden und Entfernen nicht genutzter Objekte (Garbage Collection) erschwert [46]. Offenheit, Ortstransparenz und Zugriffstransparenz sind zentrale Charakteristika verteilter objektorientierter Systeme.

Neben diesen grundlegenden Mechanismen verfügen verteilte objektorientierte Systeme noch über eine Vielzahl von Diensten. Die Auswahl der Dienste und ihre Beschaffenheit variiert in konkreten Systemen sehr stark. Im folgenden werden einige Dienste kurz beschrieben:

- Namensdienste ermöglichen es, intuitive Namen für Objekte zu vergeben. Sie ermöglichen die Abbildung dieser Namen auf Objektreferenzen.

- Sicherheitsdienste beschränken und sichern den Zugriff auf Objekte, indem Mechanismen zur Authentifikation und Verschlüsselung bereitgestellt werden.

- Dienste zur Verwaltung des Objekt-Lebenszyklus bieten Operationen zum Erzeugen und Zerstören von Objekten an. Sie bilden das verteilte Gegenstück zu den `new` und `delete` Operationen objektorientierter Programmiersprachen.

- Dienste zur Transaktionsverwaltung ermöglichen die konsistente Abarbeitung einer definierten Folge von entfernten Methodenaufrufen.

- Dienste für Objektpersistenz machen die Lebensdauer eines Objekts unabhängig von der Lebensdauer des ausführenden Servers oder Rechners. Dazu werden Mechanismen bereitgestellt, um den Zustand von Objekten dauerhaft zu speichern und zu einem späteren Zeitpunkt wieder Objekte daraus zu erzeugen.

Verteilte objektorientierte Systeme verfügen über den höchsten Abstraktionsgrad aller bisher beschriebenen Programmiermodelle. Sie heben den Abstraktionsgrad der Client-Server-Kommunikation auf das Niveau der objektorientierten Programmierung, was einen wesentlichen Vorteil gegenüber RPC-basierten Systemen darstellt. Die Verwendung von Objektreferenzen zur Adressierung von Objekten und das Konzept des entfernten Methodenaufrufs gewährleisten Orts- und Zugriffstransparenz im verteilten System. Schließlich ermöglichen eine Vielzahl von Diensten die standardisierte Behandlung häufig auftretender Problemstellungen.

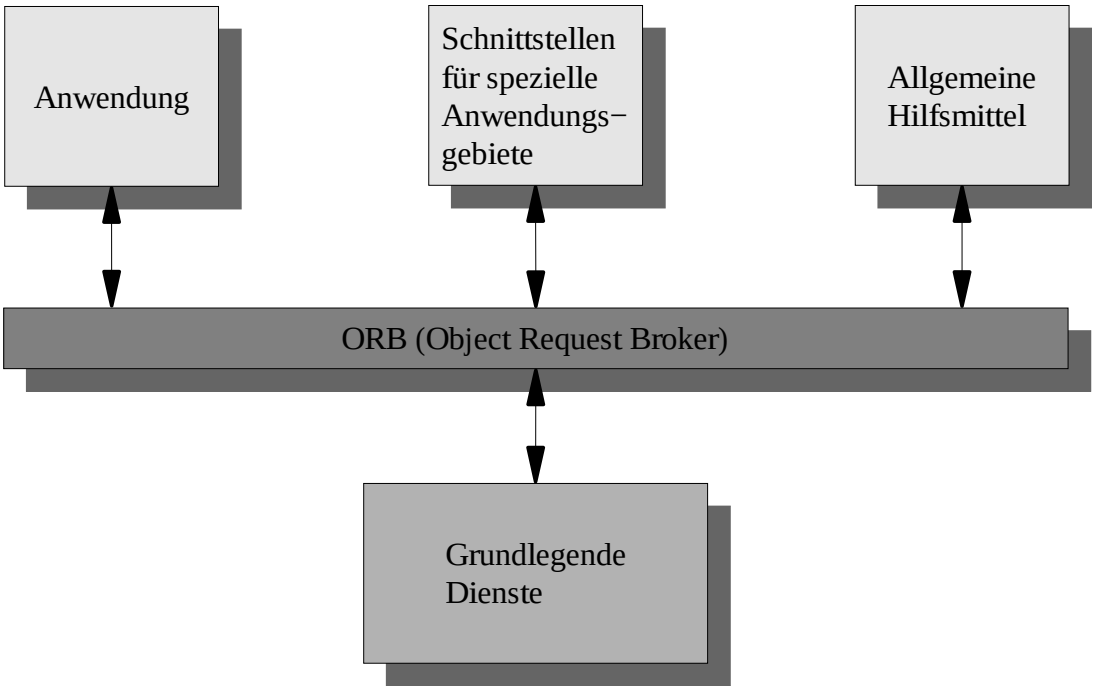
Im folgenden werden einige ausgewählte Systeme näher betrachtet. Dabei sollen ihre Eigenschaften anhand der hier vorgestellten Konzepte beschrieben werden. Im einzelnen wird auf OMG CORBA, Microsoft DCOM und Sun RMI eingegangen. Ein sehr detaillierter Vergleich dieser Systeme ist in [103, 17] zu finden.

2.4.1 CORBA

Die Common Object Request Broker Architecture (CORBA) ist ein Standard zur Entwicklung verteilter objektorientierter Anwendungen. CORBA wird von der Object Management Group (OMG) spezifiziert. Das Programmiermodell von CORBA basiert auf der Object Management Architecture (OMA) [89], welche die Struktur CORBA basierter Anwendungen beschreibt.

Abbildung 2.4 zeigt den grundlegenden Aufbau der OMA. Kern der Object Management Architecture ist der Object Request Broker (ORB) [98]. Der ORB ist eine logisch zentrale Komponente, welche die Infrastruktur zur Kommunikation bereitstellt. Die eigentliche Anwendung besteht aus einer Menge von Objekten, die über den ORB kommunizieren. Dabei steht einer Anwendung ein reichhaltiges Angebot an Diensten unterschiedlichen Abstraktionsgrades zur Verfügung. Auf unterster Ebene findet man grundlegende Dienste wie beispielsweise einen Namensdienst, einen Sicherheitsdienst oder einen Dienst zur Transaktionsverwaltung [90]. Daneben gibt es eine Reihe allgemeiner Hilfsmittel [88]. Das sind Dienste, die problemübergreifende Aufgaben wie z.B. Metaobjekte, Verbundobjekte und Agenten abdecken. Auf oberster Ebene befinden sich Schnittstellen für spezielle Anwendungsgebiete wie Telekommunikation oder Gesundheitswesen. Die Bereitstellung von Diensten unterschiedlichen Abstraktionsgrades stellt eine wesentliche Innovation gegenüber anderen Programmiermodellen

Abbildung 2.4 Die Object Management Architecture



dar. Sie leistet einen bedeutenden Beitrag zur Steigerung des Abstraktionsgrades des gesamten Programmiermodells.

Die Schnittstellen von CORBA Objekten werden, wie in verteilten objektorientierten Systemen üblich, mit Hilfe einer Beschreibungssprache definiert. CORBA IDL unterstützt die Behandlung von Ausnahmen und die Vererbung von Schnittstellen. Eine weitere Eigenschaft ist die Sprachunabhängigkeit von IDL. Dadurch kann die Spezifikation der Schnittstellen von der Implementierung der Anwendung getrennt werden. Ein IDL-Compiler erzeugt aus der Schnittstellenbeschreibung Stubs und legt zusätzlich Information über die Schnittstellen in einem Interface Repository ab. Mit Hilfe der Stubs kann der entfernte Methodenaufruf, wie er in Abschnitt 2.4 beschrieben ist, realisiert werden. Diese Art des entfernten Methodenaufrufs bezeichnet man als statisch, da die Schnittstelle des anzufragenden Objekts zum Zeitpunkt der Übersetzung bekannt sein muß. CORBA stellt auch Funktionalität für dynamische Methodenaufrufe bereit. Dabei werden Methodenaufrufe ohne Zuhilfenahme eines IDL-Compilers, mittels der Informationen aus dem Interface Repository, erzeugt. Stubs sind in diesem Fall nicht nötigt. Der Vorteil dynamischer Methodenaufrufe ist, daß die Schnittstelle des anzufragenden Objekts zum Zeitpunkt der Übersetzung nicht bekannt sein muß.

Die Adressierung von Objekten geschieht in CORBA über Objektreferenzen, die als Interoperable Object References (IOR) bezeichnet werden. Der ORB erlaubt die Umwandlung von Objektreferenzen in Zeichenketten und umgekehrt. Das ermöglicht ihre externe Speicherung und Weitergabe, erschwert aber andererseits die Garbage Collec-

tion. Objekte werden bei Methodenaufrufen als Referenzparameter behandelt. Momentan wird an einer Erweiterung des Standards gearbeitet, welche die Übergabe von Objekten als Werteparameter ermöglicht [91]. Auf der Ebene des verteilten Ablaufsystems läuft die Kommunikation über das General Inter-ORB-Protocol (GIOP). Dabei handelt es sich um ein abstraktes Protokoll auf dem konkrete Protokolle aufsetzen können. Eine weit verbreitete Implementierung von GIOP ist das Internet Inter-ORB-Protocol (IIOP), das die Kommunikation auf TCP/IP[3] abbildet. Für die Interoperabilität mit anderen verteilten Systemen wie beispielsweise DCE stehen sogenannte Environment-Specific Inter-ORB-Protocols (ESIOPs) zur Verfügung.

Die Entwicklung des CORBA Standards reicht zurück bis ins Jahr 1991 als CORBA 1.0 veröffentlicht wurde. Hauptaugenmerk war damals die IDL und ihre Abbildung auf verschiedene Programmiersprachen. Die fehlende Standardisierung des Kommunikationsprotokolls führte zu einer Vielzahl zueinander inkompatibler Implementierungen. In Version 2.0 wurden schließlich GIOP und IIOP eingeführt, was erstmals die Interoperabilität von Systemen unterschiedlicher Hersteller gewährleistete. Heute zeichnet sich CORBA insbesondere durch eine große Anzahl an freien und kommerziellen Implementierungen und ihre gute Interoperabilität aus.

2.4.2 DCOM

Die Ursprünge des Distributed Component Object Model (DCOM) [78, 24] sind im Object Linking & Embedding (OLE) zu sehen. OLE wurde von Microsoft entwickelt, um die Interaktion unabhängiger Windows-Anwendungen zu ermöglichen. Die erste Version erwies sich jedoch als zu komplex und unflexibel, woraufhin 1993 mit der Weiterentwicklung OLE 2 das Component Object Model (COM) eingeführt wurde. COM bietet einen, auf einem einheitlichen Objektmodell basierenden, Standard zur Kommunikation lokaler Objekte. Mit der Einführung von Windows NT 4.0 im Jahr 1996 wurde schließlich COM zu DCOM erweitert, das auch die Kommunikation entfernter Objekte unterstützt.

Die Schnittstellenbeschreibungssprache von DCOM ist Microsoft IDL (MIDL). Ein MIDL-Compiler erzeugt aus einer Schnittstellenbeschreibung Client- und Server-Stubs. Im Gegensatz zu CORBA, das die Generierung von Stubs für unterschiedliche Programmiersprachen erlaubt, legt DCOM das Binärformat der Schnittstellen fest. Übersetzte DCOM Schnittstellen basieren auf dem Konzept der Function Tables. Im Gegensatz zu CORBA sind also spezielle Compiler nötig, welche dieses Konzept unterstützen. Analog zum Interface Repository in CORBA können vom MIDL-Compiler sogenannte Type Libraries erzeugt werden.

COM Objekte sind keine Objekte im herkömmlichen Sinn. Vielmehr ähneln sie Komponenten, d.h. ein COM Objekt bietet nach außen hin mehrere Schnittstellen an. Statt der Vererbung bietet COM die Aggregation. Sie ermöglicht die Integration der Schnittstellen verschiedener Objekte in ein COM Objekt. Die Aggregation ist für Clients transparent. Die Adressierung erfolgt in COM über Interface Identifier (IID). Ein IID

[3] Das Transmission Control Protocol (TCP) ist ein verbindungsorientiertes Kommunikationsprotokoll das wiederum auf dem Internet Protocol (IP) aufsetzt.

ist ein eindeutiger Bezeichner für eine Schnittstelle. Da mehrere Objekte dieselbe Schnittstelle implementieren können und somit über dieselbe IID verfügen, wäre es dadurch unmöglich, eine spezielle Instanz einer COM Klasse zu adressieren. Dieses Problem wird umgangen, indem man Objekte mit zusätzlichen Identifikatoren, sogenannten Monikers, versieht. COM unterstützt sowohl statische als auch dynamische Methodenaufrufe. Die Parameter von Methodenaufrufen werden mittels der Network Data Representation (NDR), einem systemunabhängigem Datenformat, kodiert. Eine zentrale Rolle in DCOM spielt der Service Control Manager (SCM). Der SCM ist für die Erzeugung von Objekten zuständig. Dazu wird, analog zum Factory Entwurfsmuster [34], für jede COM Klasse eine Factory bereitgestellt. Der SCM lokalisiert eine Factory und erzeugt damit das entsprechende Objekt.

Das Angebot an zusätzlichen Diensten ist weniger reichhaltig als bei CORBA. DCOM bietet in Zusammenarbeit mit dem Microsoft Transaction Server (MTS) Unterstützung zur Transaktionsverwaltung. Es werden sowohl transitive Beziehungen zwischen Objekten als auch verschachtelte Transaktionen unterstützt.

Wesentliche Nachteile von DCOM sind die Verwendung von proprietären Compiler-Erweiterungen (Function Tables) und die starke Ausrichtung auf Windows. Beides erschwert die Portierung auf andere Betriebssysteme.

2.4.3 RMI

Die Remote Method Invocation (RMI) [136] ist eine Erweiterung der Programmiersprache Java, welche die Entwicklung verteilter Anwendungen ermöglicht. RMI basiert, wie auch CORBA und DCOM, auf dem Broker Entwurfsmuster.

RMI verwendet keine explizite IDL. Objekte werden als entfernt zugreifbar gekennzeichnet indem man sie von der Basisklasse `java.rmi.Remote` ableitet. Der Nachteil dieser Vorgehensweise ist die ausschließliche Bindung an die Programmiersprache Java. Andererseits wird dadurch vollständige Vererbung ermöglicht. Im Gegensatz zu CORBA, das lediglich die Vererbung von Schnittstellen unterstützt, bieten Java und RMI Vererbung auf Klassenebene. Eine Besonderheit von Java ist die Unterstützung von Objektpersistenz. Durch den Mechanismus der Objektserialisierung [135] kann der Zustand eines Objekts dauerhaft gespeichert werden, um zu einem späteren Zeitpunkt wieder ein Objekt daraus zu erzeugen. Lokale Objekte können mit Hilfe der Objektserialisierung als Werteparameter übergeben werden. Bei entfernten Objekten wird der lokale Stellvertreter in serialisierter Form übergeben. Mittels Introspektion können aus der Kopie des Stellvertreters alle Informationen für dynamische Methodenaufrufe gewonnen werden. Eine weitere Besonderheit von Java ist die virtuelle Maschine, die für alle Java Anwendungen ein homogenes Rechensystem bereitstellt. Dadurch entfällt die Kodierung der Parameter entfernter Methodenaufrufe.

RMI verfügt über einen Namensdienst, die RMI Registry, der es ermöglicht sprechende Namen für Objekte zu vergeben. Der Namensdienst ist verteilt realisiert, wobei auf jedem Rechner eine Instanz der RMI Registry läuft. Die einzelnen Instanzen werden über Uniform Resource Locators (URL) adressiert.

Wesentliche Vorteile von RMI sind die Unterstützung von Persistenz und die Vererbung auf Klassenebene. Diese Vorteile werden jedoch mit dem Nachteil erkauft, daß RMI an die Programmiersprache Java gebunden ist. Damit wird im Gegensatz zu anderen Systemen keine Heterogenität auf der Ebene der Anwendung unterstützt.

2.5 Systeme für spezielle Anwendungsgebiete

Die bisher vorgestellten Programmiermodelle sind nicht an bestimmte Anwendungsgebiete gebunden. In manchen Bereichen ist zwar eine gewisse Spezialisierung festzustellen, diese ist jedoch weniger auf die grundlegende Konzeption der Systeme, als vielmehr auf einige herausragende Eigenschaften zurückzuführen. Im folgenden werden Programmiermodelle betrachtet, die speziell für den Einsatz in eng umgrenzten Anwendungsgebieten konzipiert wurden.

2.5.1 Verteilter gemeinsamer Speicher

Im Bereich der parallelen Programmierung wird zwischen Rechnern mit gemeinsamem und verteiltem Speicher unterschieden [144]. Gemeinsamer Speicher ermöglicht es Anwendungen, über einen globalen Adreßraum zu kommunizieren. Bei verteiltem Speicher erfolgt die Kommunikation, wie in Abschnitt 2.2 beschrieben, über Nachrichtenaustausch. Programmiermodelle mit verteiltem gemeinsamen Speicher (Distributed Shared Memory, DSM) stellen dem Entwickler einen virtuellen globalen Adreßraum, in einem Rechensystem mit verteiltem Speicher, zur Verfügung. Das Ziel von DSM-Systemen ist es, für den Entwickler einen sehr hohen Grad an Abstraktion zu schaffen, indem die Verteiltheit des Rechensystems möglichst vollständig durch das Ablaufsystem verborgen wird.

Systeme mit verteiltem gemeinsamen Speicher bieten dem Entwickler Programmierbibliotheken zur Verwaltung und Nutzung eines virtuellen globalen Adreßraums. Dieser ist physisch auf das gesamte Rechensystem verteilt. Zugriffe auf physisch entfernte Adressen werden vom verteilten Ablaufsystem auf Kommunikations- und Synchronisationsmechanismen des zugrundeliegenden Rechensystems abgebildet. Für die Anwendung ist dieser Vorgang transparent, so daß der Eindruck eines globalen Maschinen-Adreßraums entsteht. Konkrete Implementierungen verwenden zur Kommunikation häufig Systeme zum Nachrichtenaustausch wie beispielsweise PVM oder MPI.

Zwei bekannte DSM-Systeme sind TreadMarks [1] und Orca [2]. Systeme mit verteiltem gemeinsamen Speicher sind in der Regel weniger effizient als Systeme mit Nachrichtenaustausch, da die Optimierung der Kommunikation auf der Ebene des verteilten Ablaufsystems und nicht auf der Ebene der Anwendung erfolgt. Das ist ein wesentlicher Unterschied zu den bisher vorgestellten Programmiermodellen. Bisher konnte der Entwickler stets zwischen lokaler und entfernter Kommunikation unterscheiden und gegebenenfalls Optimierungen durchführen. Diese Option entfällt bei Systemen mit

verteiltem gemeinsamen Speicher, was einerseits den Abstraktionsgrad erhöht aber andererseits die Effizienz verringert.

2.5.2 Verteilte transaktionsorientierte Systeme

Verteilte Transaktions-Monitore (Distributed Transaction Processing Monitors, DTPM) erweitern Client-Server-Systeme um die Fähigkeit, effiziente und zuverlässige transaktionsorientierte Anwendungen zu entwickeln [23]. Die ersten Transaktions-Monitore entstanden Mitte der 1970er Jahre. Verteilte Transaktions-Monitore bieten die Möglichkeit, Transaktionen in einem verteilten System zu verwalten, zu koordinieren und zu überwachen. Eine Transaktion ist eine Menge von Operation, die zu einer Einheit zusammengefaßt werden. Transaktionen sind durch folgende Eigenschaften bestimmt [40]:

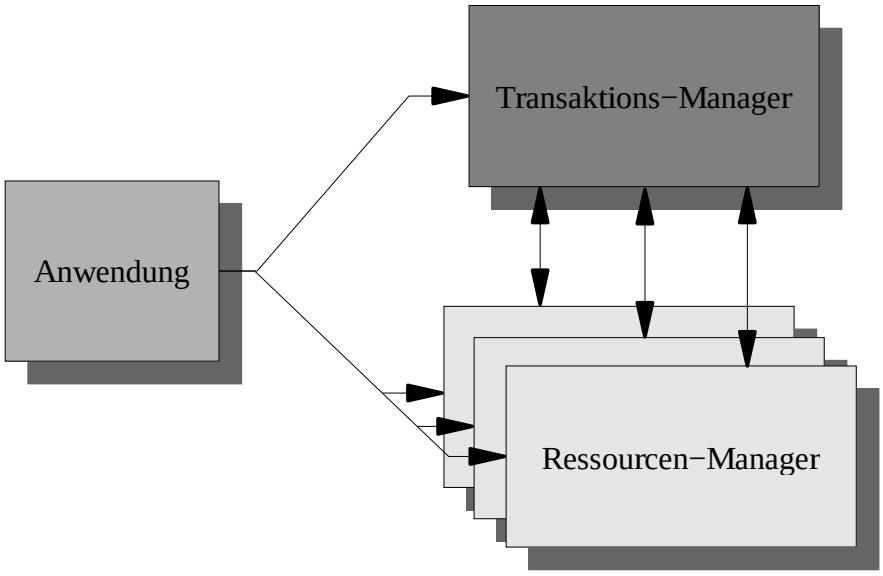
- Atomarität: Transaktionen sind die kleinsten Einheiten. Sie sind unteilbar, d.h. es werden entweder alle Operationen einer Transaktion korrekt ausgeführt oder keine.
- Konsistenz: Nach Ausführung einer Transaktion muß sich das betroffene System in einem konsistenten Zustand befinden, wenn es sich zuvor in einem konsistenten Zustand befand. Anderenfalls muß die Transaktion zurückgesetzt werden.
- Isolierung: Parallel ausgeführte Transaktionen dürfen sich nicht gegenseitig beeinflussen.
- Dauerhaftigkeit: Die Wirkung einer Transaktion bleibt dauerhaft im System erhalten. Dies muß auch bei nachfolgenden Systemfehlern gewährleistet sein.

Die wesentliche Aufgabe eines DTPM besteht darin, diese Eigenschaften zu gewährleisten und gleichzeitig für die effiziente Abarbeitung von Transaktionen zu sorgen.

Wie in Abbildung 2.5 dargestellt ist, besteht ein Transaktions-Monitor aus einem Transaktions-Manager und mehreren Ressourcen-Managern. Ressourcen-Manager statten einzelne Ressourcen wie beispielsweise Datenbanken oder Dateien mit transaktionalen Eigenschaften aus. Transaktions-Manager koordinieren Transaktionen über mehreren Ressourcen. Dies geschieht mittels des sogenannten Two-Phase Commit: Transaktionen werden in zwei Phasen aufgeteilt. In der ersten Phase werden Operationen an einzelne Ressourcen übergeben aber nicht endgültig ausgeführt. Wenn die erste Phase für alle Operationen erfolgreich war, werden die Operationen in der zweiten Phase endgültig ausgeführt. Falls die erste Phase fehlschlägt, werden alle Operationen zurückgesetzt. Der X/Open Distributed Transaction Standard [147] ist ein Referenzmodell für verteilte transaktionsorientierte Systeme. Dort werden Schnittstellen für die Interaktion von Anwendungen, Transaktions-Manager und Ressourcen-Managern definiert.

Verteilte transaktionsorientierte Systeme bilden kein eigenständiges Programmiermodell. Vielmehr sind sie als Erweiterung bestehender Systeme zu sehen, welche die

Abbildung 2.5 Schematische Darstellung eines Transaktions-Monitors



effiziente und konsistente Abarbeitung von Transaktionen ermöglicht. Deshalb sind konkrete Systeme wie beispielsweise Tuxedo von BEA sehr flexibel bezüglich der unterstützen Kommunikationsmechanismen. Sie bieten Schnittstellen für verschiedene Programmiermodelle wie z.B. Message-Queuing, RPC, DCE oder CORBA.

2.6 Zusammenfassung

In diesem Kapitel wurden grundlegende Eigenschaften verteilter Systeme, wie ihre Schichtung, die Organisationsformen und die Client-Server-Kommunikation beschrieben. Anhand dieser Eigenschaften wurden verschiedene Programmiermodelle mit ihren technischen und historischen Merkmalen vorgestellt. Besondere Beachtung fand dabei der Abstraktionsgrad der Programmiermodelle. Er beschreibt die Sichtweise des Entwicklers auf das verteilte System und ist deswegen zu einem gewissen Teil nur subjektiv erfaßbar. Um dennoch eine möglichst objektive Darstellung zu erhalten, ist es nötig, eine Unterteilung in mehrere Merkmale vorzunehmen. Geeignete Merkmale sind der Kommunikationsmechanismus, die verfügbaren Dienste und die Schichtungstiefe des Programmiermodells. In Tabelle 2.1 werden einige ausgewählte Programmiermodelle anhand dieser Merkmale verglichen. Der Abstraktionsgrad des Programmiermodells ergibt sich aus dem Mittelwert der einzelnen Merkmale.

Unter den Kommunikationsmechanismen bietet der entfernte Methodenaufruf den höchsten Abstraktionsgrad, gefolgt vom entfernten Prozeduraufruf und der nachrichtenorientierten Kommunikation. Hinsichtlich der verfügbaren Dienste schneidet der RPC am schlechtesten ab. Message Oriented Middleware bietet lediglich grundlegende Diente wie z.B. Ausfallsicherheit im Bereich der Message-Queuing-Systeme und

Tabelle 2.1 Gegenüberstellung verschiedener Programmiermodelle

	MOM	RPC	DCE	CORBA
Kommunikation	*	**	**	***
Dienste	**	*	***	***
Schichtung	*	*	**	***
Abstraktionsgrad	*	*	**	***

* schlecht
** gut
*** sehr gut

dynamische Prozeßverwaltung im Bereich der Message-Passing-Systeme. Das umfangreichste Angebot an Diensten bieten DCE und CORBA. Die Schichtungstiefe ist bei Message Oriented Middleware und RPC am geringsten, da nur Dienste zur Verfügung stehen. DCE bietet bereits eine Menge hierarchisch strukturierter Dienste und CORBA unterscheidet zusätzlich zwischen grundlegenden Diensten, allgemeinen Hilfsmitteln und Schnittstellen für spezielle Anwendungsgebiete. Zusammenfassend kann man sagen, daß Message Oriented Middleware und RPC über den geringsten Abstraktionsgrad verfügen, gefolgt von DCE und CORBA.

Die stetige Steigerung des Abstraktionsgrades der Programmiermodelle ist auf den wachsenden Umfang und die steigende Komplexität verteilter Anwendungen zurückzuführen. Hochstehende Kommunikationsmechanismen ermöglichen die nahtlose Integration der Kommunikation in Anwendungen. Ein umfangreiches Angebot an Diensten erlaubt häufig auftretende Probleme auf eine standardisierte Art und Weise zu lösen. Mit dem steigenden Abstraktionsgrad wächst aber auch der Umfang und die Komplexität der Programmiermodelle. Die Schichtung der Programmiermodelle und ihrer Dienste ist nötig, um ihre Handhabbarkeit weiterhin zu gewährleisten. Mit wachsendem Abstraktionsgrad steigt auch der Mehraufwand, der durch das Programmiermodell verursacht wird, was wiederum die Performanz der Anwendungen beeinträchtigt. Aus diesem Grund muß bei der Wahl eines geeigneten Programmiermodells stets zwischen dem Abstraktionsgrad und den Kosten abgewogen werden. Somit wird die Vielfalt der Programmiermodelle wohl auch in Zukunft erhalten bleiben.

3. Lastverwaltung

Ein verteiltes Rechensystem stellt eine große Menge unterschiedlicher Ressourcen, einen sogenannten Ressourcen-Pool, für Anwendungen zur Verfügung. Die Zusammensetzung des Ressourcen-Pools kann sich zur Laufzeit verändern, indem neue Ressourcen zum System hinzugefügt oder bestehende entfernt werden. Jede verteilte Anwendung nutzt einen bestimmten Teil des Ressourcen-Pools, um Aktivitäten auszuführen. Dabei verändert sich ständig die Nachfrage nach Ressourcen, da verschiedene Aktivitäten die einzelnen Ressourcen unterschiedlich stark beanspruchen. Dadurch kann es zu einer ungleichmäßigen Auslastung des Ressourcen-Pools kommen, was wiederum dazu führen kann, daß Ressourcen überlastet sind, während gleichwertige Ressourcen unterlastet sind, oder im Extremfall gar nicht genutzt werden. Dies wirkt sich negativ auf das Laufzeitverhalten verteilter Anwendungen aus. Die Lastverwaltung wirkt diesem Problem entgegen, indem sie versucht, eine gleichmäßige Auslastung des Ressourcen-Pools zu erreichen.

Das Problem der Lastverwaltung wurde bereits für viele Programmiermodelle und Anwendungen untersucht [102, 61, 114, 4, 139, 18]. Im Lauf der Zeit ist dadurch eine reichhaltige Begriffswelt entstanden; eine einheitliche Terminologie hat sich jedoch nur in bestimmten Teilbereichen der Lastverwaltung etabliert. So hat LUDWIG [71], basierend auf den Arbeiten von CASAVANT und KUHL [15], eine detaillierte Klassifikation von Lastverwaltungssystemen erstellt, deren Schwerpunkt auf der Lastbewertung liegt. RÖDER [109] widmete sich besonders der Lasterfassung und der Kategorisierung dieses Teilbereichs. Mit den Mechanismen der Lastverschiebung und insbesondere der Prozeßmigration beschäftigt sich wiederum die Arbeit von STELLNER [118]. Neben der Fokusierung auf bestimmte Teilbereiche der Lastverwaltung ist ein weiterer Grund für die Vielzahl der Klassifikationsmodelle in der ständigen Veränderung und Weiterentwicklung der Rechensysteme und ihrer Programmiermodelle zu sehen. Dadurch sind sowohl die Anforderungen als auch die Möglichkeiten der Lastverwaltung einem stetigen Wandel unterworfen, die eine Anpassung der bestehenden Klassifikationsmodelle erfordern.

Ziel dieses Kapitels ist es, die umfangreiche Terminologie aus dem Bereich der Lastverwaltung zu strukturieren, um ein Begriffsgerüst für die nachfolgenden Abschnitte

zu schaffen. Dazu wird zunächst die Aufgabenstellung der Lastverwaltung mit Hilfe der Terminologie der Regelungstechnik in logische Komponenten zerlegt. Diese Komponentenstruktur bildet die Grundlage eines neuen Klassifikationsmodells, das auf dem Begriffsgerüst der erwähnten Arbeiten basiert und dieses an kritischen Stellen erweitert und modifiziert, um den neu hinzugekommenen Anforderungen und Möglichkeiten der Lastverwaltung zu genügen. Das Klassifikationsmodell ist so strukturiert, daß damit sowohl bestehende Lastverwaltungssysteme kategorisiert als auch neue Systeme spezifiziert werden können. Dieser Aspekt ist von besonderer Bedeutung, da das Klassifikationsschema im nachfolgenden Kapitel zur Entwicklung eines neuen Lastverwaltungskonzepts herangezogen wird. Das Kapitel schließt mit einem Überblick über das Klassifikationsmodell und einigen ausgewählten Beispielen.

3.1 Grundlagen

Zunächst werden einige allgemeine Begriffe aus dem Bereich der Lastverwaltung erläutert, um somit ein Begriffsgerüst für die folgenden Abschnitte zu schaffen.

3.1.1 Grundlegende Begriffe

In einem verteilten System gibt es Leistungseinheiten, die sogenannten Ressourcen[1], die Dienstleistungen zur Verfügung stellen, und es gibt Verbraucher, welche diese in Anspruch nehmen.

Definition 3.1 *Ressource*

> Eine Ressource ist eine Komponente, die Leistung zur Verfügung stellt, damit ein Verbraucher bestimmte Aktivitäten ausführen kann. [73] □

Ressourcen findet man auf allen Ebenen eines verteilten Systems, z.B. der Prozessor und das Kommunikationsnetzwerk im Rechensystem, Betriebssystemdienste im Ablaufsystem und Objekte auf der Ebene der Anwendung. Neben diesen elementaren Ressourcen können auch größere Einheiten eines verteilten Systems als Ressourcen betrachtet werden. Dies könnten beispielsweise Rechner auf der Ebene des Rechensystems und Server auf der Ebene der Anwendung sein. Die Wahl geeigneter Ressourcen hängt letztendlich von den Anforderungen der Verbraucher und dem Detailwissen ab, das für die Lastverwaltung erforderlich ist.

Ressourcen sind entweder exklusiv oder parallel benutzbar. Bei parallel benutzbaren Ressourcen können beliebig viele Verbraucher simultan auf eine Ressource zugreifen. Exklusiv benutzbare Ressourcen dürfen zu einem Zeitpunkt nur von einem Verbraucher genutzt werden. Mehrere Verbraucher greifen dann im time-sharing-Verfahren auf die Ressourcen zu. Dabei erhält jeder Verbraucher für eine bestimmte Zeitspanne exklusive Nutzungsrechte an einer Ressource.

[1] In der deutschsprachigen Literatur werden Ressourcen häufig auch als Betriebsmittel bezeichnet.

Ressourcen stellen Leistung zur Verfügung, die von Verbrauchern konsumiert wird; in diesem Zusammenhang spricht man von der Kapazität einer Ressource. Betrachtet man beispielsweise ein Speichermedium, so besteht seine Leistung in der Bereitstellung von Speicherplatz. Die Kapazität eines Speichermediums wird in Byte gemessen. Ein Prozessor arbeitet die Instruktionen einer oder mehrerer Anwendungen ab. Die Kapazität eines Prozessors könnte beispielsweise durch die Anzahl der Instruktionen beschrieben werden, die er pro Zeiteinheit abarbeiten kann (MIPS, Millionen Instruktionen pro Sekunde) [144]. Die Leistungsfähigkeit und damit auch die maximale Kapazität jeder Ressource ist begrenzt. Deswegen sinkt die freie Kapazität mit steigender Nutzung der Ressource. Für die Lastverwaltung ist ein quantitatives Maß dieser Nutzung unerläßlich. Dieses Maß wird als Last bezeichnet:

Definition 3.2 *Last*

> Die Last einer Ressource zu einem bestimmten Zeitpunkt ist das Verhältnis der nachgefragten Kapazität und der maximal verfügbaren Kapazität der Ressource. [73] □

Bei einer Last von eins ist das Verhältnis von nachgefragter und maximal verfügbarer Kapazität ausgeglichen. Ist die Last kleiner als eins, so spricht man von Unterlast, da ein Teil der Kapazität der Ressource nicht genutzt wird. Eine Last von mehr als eins bezeichnet man als Überlast, da die Nachfrage das Angebot übersteigt. Bei Überlast kann die Nachfrage nach einer Ressource nicht sofort befriedigt werden, was zu Wartezeiten bei den Verbrauchern führt. In der Literatur findet man neben dem Begriff “Last” auch die Begriffe “Lastwert” und “Lastindex” [73, 109]. Ein Lastwert oder Lastindex ist dabei ein numerisches Maß zur groben Abschätzung der Last einer Ressource. Diese Unterscheidung wird getroffen, da die exakte Last einer Ressource häufig nur schwer meßbar ist. Hier wird bewußt auf diese Unterscheidung verzichtet, da eine exakte Betrachtung der Last für innovative Verfahren der Lastverwaltung unverzichtbar ist. Im folgenden werden die Begriffe “Lastwert” und “Last”gleichbedeutend verwendet.

In der Literatur werden für die Lastverwaltung verschiedene Ziele und Aufgabenstellungen genannt, was größtenteils auf die sehr unterschiedlichen Anwendungsgebiete zurückzuführen ist:

- Verbesserung der Antwortzeiten: Die Lastverwaltung kann dazu beitragen, die mittleren Bedien- und Wartezeiten herabzusetzen und somit die Antwortzeiten der betrachteten Ressourcen zu verbessern [114].

- Verbesserung des Durchsatzes: Mit Hilfe der Lastverwaltung kann die Arbeitslast so unter den einzelnen Ressourcen eines verteilten Systems aufgeteilt werden, daß sich dadurch der Durchsatz des Gesamtsystems erhöht.

- Erhöhung der Skalierbarkeit: Die Leistungsfähigkeit eines verteilten Systems sollte durch eine Vergrößerung des Ressourcen-Pools in vergleichbarem Maße steigen.

- Geringe Kosten der Lastverwaltung: Die Aktionen, die von einem Lastverwaltungssystem ausgeführt werden, sollten so billig als möglich sein, um den erwarteten Gewinn nicht durch den Mehraufwand, den diese Aktionen verursachen, wieder zunichte zu machen.

- Fehlertoleranz: In letzter Zeit geht man immer mehr dazu über als ein Zielkriterium der Lastverwaltung auch Fehlertoleranz zu fordern [102, 73]. Die Lastverwaltung hat die Möglichkeit Ausfälle einzelner Ressourcen zu erkennen und den betroffenen Verbrauchern alternative Ressourcen anzubieten.

Bei der Vielzahl an möglichen Zielsetzungen muß stets berücksichtigt werden, daß einige Ziele im Widerspruch stehen. Den unterschiedlichen Zielvorstellungen stehen wiederum die Möglichkeiten der Lastverwaltung gegenüber:

Definition 3.3 *Lastverwaltung*

Lastverwaltung beschäftigt sich mit dem Problem, die Arbeitslast in einem verteilten System so zwischen den einzelnen Ressourcen aufzuteilen, daß vorgegebene Ziele und Randbedingungen erfüllt werden. [73] □

Aus den Möglichkeiten und den allgemeinen Zielvorstellungen können konkrete Ziele für die Lastverwaltung abgeleitet werden. LUDWIG [71] nennt dabei zwei Ziele:

Definition 3.4 *Lastausgleich und Lastbalancierung*

Lastausgleich ist jene Zielvorstellung der Lastverwaltung, bei der gefordert wird, daß die Last von Ressourcen insofern ausgeglichen ist, daß keine Ressource unterlastet sein darf, während eine vergleichbare Ressource überlastet ist.

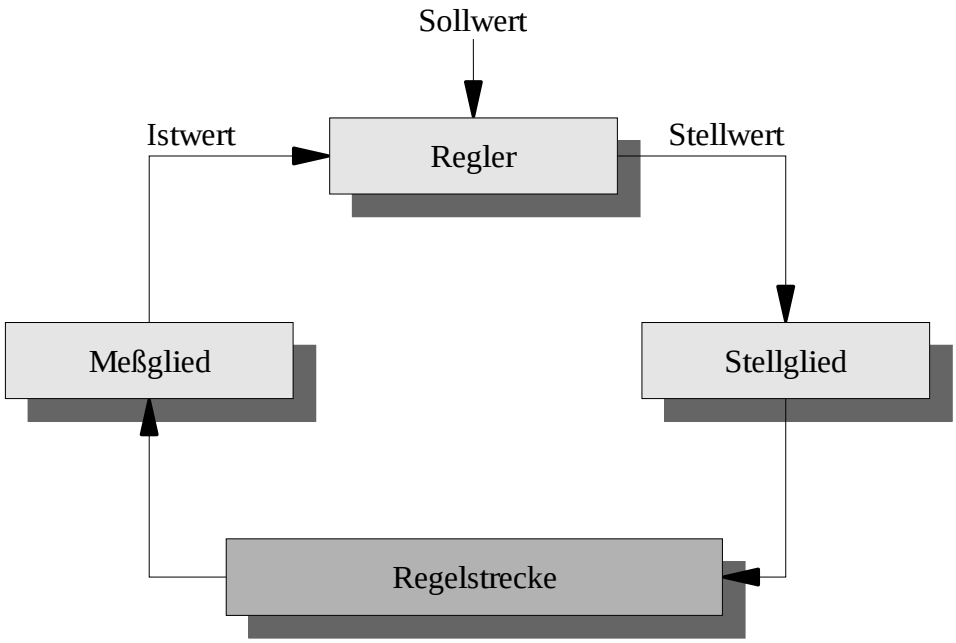
Bei der Lastbalancierung wird gefordert, daß sich die Lasten vergleichbarer Ressourcen höchstens um ein quantitatives Maß unterscheiden. □

Der Lastausgleich gewährleistet, daß keine Ressource unterlastet ist, während eine gleichwertige Ressource überlastet ist. Damit ist sichergestellt, daß die Kapazität des Ressourcen-Pools so gut wie möglich ausgenutzt wird, was man auch als Maximierung der Systemnutzung bezeichnet. Der Lastausgleich ist also ein hinreichendes Ziel zur Maximierung der Systemnutzung und stellt somit das Minimalziel der Lastverwaltung dar. Aus diesem Grund befriedigt der Lastausgleich auch allgemeine Zielvorstellungen, wie die Verbesserung der Antwortzeit, des Durchsatzes und die Erhöhung der Skalierbarkeit. Die Lastbalancierung hingegen versucht die Lastunterschiede zwischen vergleichbaren Ressourcen möglichst gering zu halten. Durch die Verwendung eines geeigneten Balancierungskriteriums kann erreicht werden, daß die Lastbalancierung auch den Lastausgleich gewährleistet. In der Praxis ist die Lastbalancierung jedoch nicht immer das Mittel der Wahl, wie die Untersuchungen von KREMIEN und KRAMER [62] zeigen. Für bestimmte Lastverhältnisse steigen die Kosten zur Realisierung der Lastbalancierung überproportional gegenüber dem erzielbaren Nutzen.

3.1.2 Komponentenstruktur eines lastverwalteten Systems

Die Lastverwaltung greift steuernd in ein verteiltes System ein, um die Arbeitslast zwischen den einzelnen Ressourcen aufzuteilen. Diese Aufgabenstellung hat einen starken Bezug zur Regelungstechnik [77]. Um eine bessere Strukturierung der Lastverwaltung zu ermöglichen, wird zuerst ein Regelkreis betrachtet wie er in Abbildung 3.1 dargestellt ist. Eine Regelung kümmert sich um die Einhaltung gewünschter Be-

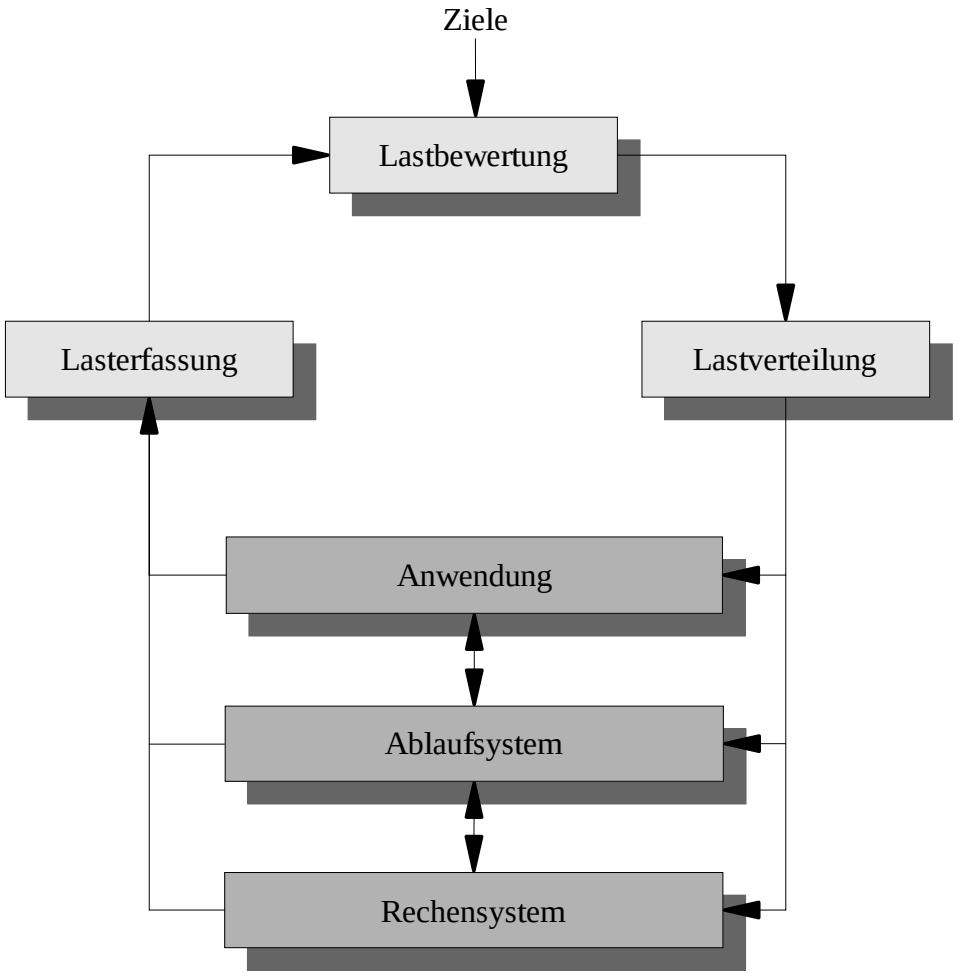
Abbildung 3.1 Der Regelkreis



dingungen in einem betrachteten System, der sogenannten Regelstrecke. Das Meßglied ist eine Komponente, die den aktuellen Systemzustand beobachtet und Meßwerte (Istwert) an den Regler liefert. Der Regler vergleicht den Istwert mit einem vorgegebenen Sollwert und ermittelt mit Hilfe eines bestimmten Algorithmus einen Stellwert. Das Stellglied greift dann, entsprechend dem Stellwert, verändernd in die Regelstrecke ein. Diese Veränderung löst eine Rückkopplung aus, die wiederum vom Meßglied erfaßt wird.

Mit Hilfe des Regelkreises kann nun die Lastverwaltung in logische Komponenten zerlegt werden. Die Struktur eines lastverwalteten Systems ist in Abbildung 3.2 dargestellt. Die Regelstrecke ist in diesem Fall ein verteiltes System. Die Lasterfassung entspricht dem Meßglied und liefert Lastinformation über die verschiedenen Ebenen des verteilten Systems an die Lastbewertung. Die Lastbewertung benutzt diese Information um entsprechend der vorgegebenen Ziele eine Entscheidung über die Verteilung von Arbeitslast zu treffen. Analog zum Stellglied ist die Lastverteilung dafür verantwortlich, die Entscheidungen der Lastbewertung umzusetzen. Dazu greift sie auf unterschiedlichen Ebenen des verteilten Systems steuernd ein. Da die einzelnen Ebe-

Abbildung 3.2 Komponentenstruktur eines lastverwalteten Systems



nen in einer engen Beziehung zueinander stehen, beeinflußt die Lastverteilung direkt oder indirekt alle Ebenen des verteilten Systems. Der Kreislauf schließt sich, indem die Lasterfassung die Auswirkungen der Lastverteilung zur Kenntnis nimmt.

Die Lastverwaltung kann also, in Anlehnung an verwandte Problemstellungen aus der Regelungstechnik, in drei logische Komponenten unterteilt werden.

Definition 3.5 *Lasterfassung, Lastbewertung und Lastverteilung*

Die Aufgabe der Lasterfassung ist es, Lastinformation auf allen relevanten Ebenen eines verteilten Systems zu messen und zu sammeln.

Die Lastbewertung nutzt die Lastinformation, um anhand vorgegebener Ziele, eine Entscheidung über die Verteilung der Arbeitslast zu treffen.

Die Lastverteilung greift aktiv in ein verteiltes System ein, um eine Verteilung der Arbeitslast zu bewirken. □

Die Aufspaltung in logische Komponenten ermöglicht eine differenzierte Betrachtung und eine getrennte Klassifikation der einzelnen Funktionseinheiten.

3.2 Klassifikation von Lastverwaltungssystemen

Dank der intensiven Forschung der vergangenen Jahre ist im Bereich der Lastverwaltung eine sehr umfangreiche aber leider auch unstrukturierte Begriffswelt entstanden. Die bisher veröffentlichten Klassifikationsmodelle versuchen entweder die Lastverwaltung als Ganzes zu kategorisieren oder sie beschränken sich auf einen speziellen Teilbereich. Die ausschließliche Betrachtung einzelner Bereiche führt zu Überschneidungen und Inkonsistenzen der Terminologie. Die Kategorisierung der Lastverwaltung als Ganzes ist wegen des großen Umfangs schwierig und verleitet wiederum dazu, sich zu sehr auf bestimmte Teilbereiche zu konzentrieren. Die in Abschnitt 3.1.2 hergeleitete Komponentenstruktur ermöglicht es, die einzelnen Funktionseinheiten differenziert zu betrachten und gleichzeitig ihre Einordnung in den Gesamtkontext der Lastverwaltung zu gewährleisten.

3.2.1 Lasterfassung

Die Lasterfassung mißt und sammelt Information über die Auslastung von Ressourcen. Die einzelnen Ressourcen können verschiedenen Schichten eines verteilten Systems zugeordnet werden. Die betrachteten Schichten werden als Ebenen der Lasterfassung bezeichnet. Sie beschreiben die Sichtweise der Lastverwaltung auf das verteilte System.

Definition 3.6 *Ebenen der Lasterfassung*

> Als Ebenen der Lasterfassung bezeichnet man alle Schichten eines verteilten Systems, die bei der Lasterfassung betrachtet werden. Dies können das Rechensystem, das Ablaufsystem und die Anwendung sein. □

Um die Lastinformation und ihre Beschaffenheit konkret beschreiben zu können, müssen die einzelnen Ressourcen der Lasterfassung betrachtet werden:

Definition 3.7 *Ressourcen der Lasterfassung*

> Als Ressourcen der Lasterfassung bezeichnet man alle Ressourcen eines verteilten Systems, die bei der Lasterfassung betrachtet werden. □

Häufig verwendete Ressourcen des Rechensystems sind beispielsweise der Prozessor, der Speicher und das Kommunikationsnetzwerk. Auf der Ebene des Ablaufsystems findet man Prozesse, Threads und Betriebssystemdienste, und auf der Ebene der Anwendung Programme, Dienste und Objekte. Je nach Kontext sind auch weitere Ressourcen vorstellbar.

Neben diesen eher konzeptionellen Aspekten kann auch die technische Umsetzung der Lasterfassung als Klassifikationsmerkmal herangezogen werden. In diesem Zusammenhang ist die Integration der Lasterfassung von besonderer Bedeutung.

Definition 3.8 *Integration der Lasterfassung*

> Die Lasterfassung kann auf verschiedenen Ebenen eines verteilten Systems realisiert werden. Sie kann in das Rechensystem, das Ablaufsystem und die Anwendungen integriert sein. □

Die Integration der Lasterfassung in das Rechensystem erfordert sogenannte Hardwaremonitore, die über elektronische Meßfühler die Last an bestimmten Meßpunkten eines Rechners oder eines Kommunikationsnetzwerks registrieren. Hardwaremonitore sind minimal invasiv, da sie selbst keine Ressourcen des Rechensystems beanspruchen. Das verteilte System wird durch die Messung nicht zusätzlich belastet, was wiederum zu exakten Meßwerten führt. Hardwaremonitore werden jedoch nur selten eingesetzt, da sie in handelsüblichen Rechnern nur in sehr begrenztem Umfang enthalten sind. Neben dem Rechensystem kann die Lasterfassung auch in das Ablaufsystem integriert sein. Dies geschieht entweder durch Instrumentierung oder durch die Nutzung expliziter Schnittstellen des Ablaufsystems. UNIX-Betriebssysteme bieten eine Vielzahl von Schnittstellen an, die Lastinformation auf unterschiedlichen Ebenen liefern. Beispielsweise beschreibt der sogenannte `avenrun`-Wert die Prozessorlast anhand der durchschnittlichen Anzahl rechenbereiter Prozesse. Die Integration der Lasterfassung in die Anwendung erfolgt stets mittels Instrumentierung. Häufig werden dabei Schleifenzähler oder Leerlaufzeiten der Anwendung betrachtet.

Wie an diesen Beispielen zu sehen ist, entscheidet die Integration der Lasterfassung zu einem großen Teil darüber, welche Lastwerte gemessen werden können. Ist die Lasterfassung in das Rechensystem integriert, so kann sie nur wenig Lastinformation über die Anwendung liefern; ist sie in die Anwendung integriert, so kann nur schwer Lastinformation über das Rechensystem gewonnen werden. Die Integration in das Ablaufsystem hingegen erlaubt es, Information auf allen Ebenen eines verteilten Systems zu sammeln, da das Ablaufsystem als Bindeglied zwischen dem Rechensystem und der Anwendung dient.

In der Literatur werden häufig die Begriffe “anwendungs-” und “systemintegrierte Lastverwaltung” verwendet. Diese Unterscheidung ist aus zwei Gründen ungenau: Zum einen kann die Integration nicht nur auf der Ebene der Anwendung und des Betriebssystems sondern auch auf der Ebene des Rechensystems erfolgen. Zum anderen muß unterschieden werden, um welche Komponente der Lastverwaltung es sich handelt. Sowohl die Lasterfassung als auch die Lastverteilung können auf unterschiedlichen Ebenen eines verteilten Systems realisiert werden[2]. Es ist sogar denkbar, daß eine Komponente anwendungsintegriert und eine andere systemintegriert ist.

[2]Wenn alle Komponenten auf der Ebene der Anwendung realisiert sind, dann handelt es sich nicht mehr um ein Lastverwaltungssystem, sondern um eine Anwendung, die Lastverwaltung betreibt [76, 127]. Diese Klasse von Anwendungen wird hier nicht betrachtet.

Ein selten erwähntes aber dennoch wichtiges Merkmal der Lasterfassung ist die Transparenz. Sie beschreibt den Mehraufwand, der für den Entwickler einer verteilten Anwendung durch die Lasterfassung entsteht.

Definition 3.9 *Transparenz der Lasterfassung*

> Die Lasterfassung ist transparent, wenn dadurch für den Entwickler einer verteilten Anwendung kein Mehraufwand entsteht. Anderenfalls ist sie nicht-transparent. □

Die Transparenz hängt teilweise von der Integration der Lasterfassung ab. Die Integration auf der Ebene des Rechen- und des Ablaufsystems ist in der Regel transparent, da dadurch kein Mehraufwand für den Entwickler entsteht. Bei der Integration in die Anwendung muß unterschieden werden, ob die Instrumentierung manuell erfolgt oder ob dazu ein automatisches Werkzeug, wie z.B. ein Compiler, verwendet wird.

3.2.2 Lastbewertung

Die Lastbewertung nutzt die Information der Lasterfassung um anhand vorgegebener Ziele eine Entscheidung über die Verteilung von Arbeitslast zu treffen. Diese Komponente beinhaltet die eigentliche Intelligenz der Lastverwaltung und wurde deshalb schon in vielen wissenschaftlichen Arbeiten untersucht [71, 118, 109, 62]. Die hier vorgestellte Klassifikation umfaßt die Struktur und den Aufbau der Lastbewertung; Algorithmen und Strategien behandelt Kapitel 5.

Die Lastbewertung ist, wie in Abschnitt 3.1.2 beschrieben, eine logisch zentrale Komponente. Die logische Struktur hat jedoch keinen Einfluß auf die Implementierung. Es ist sowohl eine zentrale als auch eine verteilte Realisierung dieser Komponente vorstellbar. Die örtliche Anordnung der Komponenten bezeichnet LUDWIG [71] als Lokalisation der Lastbewertung. Im Gegensatz zur ursprünglichen Definition, die sich auf Rechner als Verteilungseinheiten festlegt, werden hier ganz allgemein Ressourcen betrachtet.

Definition 3.10 *Lokalisation der Lastbewertung*

> Unter einer lokalen Lastbewertungskomponente versteht man eine Einheit, die in äquivalenter Form für jede betrachtete Ressource existiert.
>
> Eine Lastbewertungskomponente bezeichnet man als gruppiert, wenn für je eine Gruppe von Ressourcen eine solche Komponente vorhanden ist und insgesamt mehrere solcher Einheiten existieren.
>
> Eine zentrale Lastbewertungskomponente existiert im verteilten System nur in einer einzigen Ausprägung. [71] □

Die Lokalisation der Lastbewertung beschreibt lediglich die örtliche Anordnung der einzelnen Komponenten. Die Zuständigkeit für bestimmte Ressourcen wird als Wirkungsbereich bezeichnet:

Definition 3.11 *Wirkungsbereich der Lastbewertung*

Als Wirkungsbereich der Lastbewertung bezeichnet man denjenigen Teil des verteilten Systems, innerhalb dessen diese Komponente Lastverwaltung durchführen kann.

Ein minimaler Wirkungsbereich umfaßt lediglich zwei Ressourcen.

Ein begrenzter Wirkungsbereich liegt vor, wenn die Lastbewertung einen Teil des Systems verwaltet, der aus mehr als zwei, aber nicht aus allen Ressourcen besteht.

Von einem maximalen Wirkungsbereich spricht man, wenn eine Lastbewertungskomponente auf das gesamte System einwirken kann. □

Die Unterscheidung zwischen Lokalisation und Wirkungsbereich ist nötig, da es Lastbewertungskomponenten gibt, deren Wirkungsbereiche überlappen. In diesem Fall kann nicht vom Wirkungsbereich auf die Lokalisation geschlossen werden.

Je nach Lokalisation kann es mehrere Lastbewertungskomponenten in einem System geben. Deshalb muß unterschieden werden, ob und in welcher Form diese zusammenarbeiten.

Definition 3.12 *Interaktion der Lastbewertungskomponenten*

Ein System mit mehreren Lastbewertungskomponenten bezeichnet man als kooperierend, wenn zwischen mindestens zwei solcher Komponenten ein Informationsaustausch erfolgt und die ausgetauschte Information zur Entscheidungsfindung verwendet wird.

Ist dies nicht der Fall, d.h. arbeitet jede Komponente für sich alleine, so nennt man sie autonom. [71] □

Kooperierende Lastbewertungskomponenten müssen ihre Entscheidungen aufeinander abstimmen. LUDWIG [71] beschreibt zwei Möglichkeiten wie diese Abstimmung erfolgen kann:

- Konsensfindung: Alle Komponenten führen die Lastbewertung durch und gelangen durch Abstimmung zum Konsens.

- Arbeitsteilung: Die Komponenten teilen die unterschiedlichen Arbeitsschritte unter sich auf. Beispielsweise führen alle Komponenten die Lastbewertung durch und eine ausgezeichnete Komponente trifft anschließend die Entscheidung über die Verteilung der Last.

Als weiteres Klassifikationsmerkmal nennt LUDWIG [71] die Systemkenntnis. Sie beschreibt die Sichtweise der Lastbewertung auf das verteilte System. Auch an dieser Stelle wird die ursprüngliche Definition insofern verallgemeinert, daß Ressourcen statt Rechner verwendet werden.

Definition 3.13 *Systemkenntnis der Lastbewertung*

Von lokaler Systemkenntnis spricht man, wenn eine Lastbewertungskomponente lediglich Information über eine einzelne Ressource hat.

Bei partieller Systemkenntnis verfügen die Lastbewertungskomponenten über Lastinformation von mehreren aber nicht von allen Ressourcen.

Von globaler Systemkenntnis spricht man, wenn eine Lastbewertungskomponente Kenntnis über alle Ressourcen des Systems hat. □

Je umfassender die Systemkenntnis, desto fundierter kann die Entscheidung der Lastbewertung sein. Mit wachsender Systemkenntnis steigt jedoch auch der Aufwand zur Auswertung der Lastinformation, was die Kosten der Lastverwaltung erhöht. In der Praxis muß also zwischen den Kosten und der Qualität der Entscheidungsfindung abgewogen werden.

Ein weiteres Klassifikationsmerkmal ist die Informationsgrundlage der Lastbewertung. Sie legt fest, ob zur Entscheidungsfindung überhaupt Information über den Systemzustand herangezogen wird oder nicht [64].

Definition 3.14 *Informationsgrundlage der Lastbewertung*

Die Informationsgrundlage der Lastbewertung ist dynamisch, wenn zur Entscheidungsfindung auf Information über den Zustand des verteilten Systems zurückgegriffen wird.

Bei statischer Informationsgrundlage arbeitet die Lastbewertung ohne Kenntnis des Systemzustandes. □

Verfahren mit statischer Lastbewertung folgen nicht dem in Abschnitt 3.1.2 vorgestellten Regelkreisschema, da die Lasterfassungskomponente entfällt. Lastausgleich kann damit nur gewährleistet werden, wenn entweder die Ressourcenanforderungen aller Verbraucher identisch sind oder nur überlastete Ressourcen auftreten. Anderenfalls ist die statische Lastbewertung nicht in der Lage, das Minimalziel der Lastverwaltung, den Lastausgleich, zu erreichen. Statische Lastbewertung wird häufig auf der Ebene des Rechensystems eingesetzt, beispielsweise in Netzwerk-Routern [18]. In diesem Fall kann die Lastverteilung ausschließlich anhand von Schätzverfahren erfolgen.

In der Literatur werden häufig die Begriffe “dynamisch” und “adaptiv” gleichgesetzt [25]. Einige Arbeiten sehen jedoch in der Adaptivität der Lastbewertung die Fähigkeit, sich an Veränderungen im verteilten System anzupassen, die außerhalb der Kontrolle der Lastverwaltung liegen [118]. In Anlehnung daran wird hier die folgende Definition verwendet:

Definition 3.15 *Adaptivität der Lastbewertung*

Die Lastbewertung bezeichnet man als adaptiv, wenn sie in der Lage ist, sich an eine Veränderung der Systemlast anzupassen, die durch Verbraucher hervorgerufen wurde, die nicht der Kontrolle der Lastverwaltung unterliegen. Anderenfalls ist sie nicht-adaptiv. □

Adaptive Lastbewertung kommt insbesondere dann zum Einsatz, wenn mehrere Anwendungen in einem System betrieben werden, von denen nicht alle der Lastverwaltung unterliegen. Die nicht lastverwalteten Anwendungen erzeugen sogenannte Hintergrundlast, die von der Lastverwaltung ausgeglichen werden muß. Die anfangs erwähnte Unterscheidung der Begriffe “dynamisch” und “adaptiv” ist durchaus sinnvoll, da adaptive Verfahren zwar stets dynamisch sind, aber dynamisch Verfahren nicht unbedingt adaptiv sein müssen.

Schließlich muß bei der Lastverwaltung noch unterschieden werden, wer die Lastbewertung steuert und initiiert.

Definition 3.16 *Steuerung und Initiierung der Lastbewertung*

Man spricht von lastgesteuerter Lastbewertung, wenn das Durchlaufen des Regelkreises aufgrund von überlasteten und/oder unterlasteten Ressourcen angestoßen wird.

Die Lastbewertung wird als überlastgesteuert bezeichnet, wenn sie durch Überlast ausgelöst wird.

Im Gegensatz dazu spricht man von unterlastgesteuerter Lastbewertung, wenn sie aufgrund von unterlasteten Ressourcen angestoßen wird.

Die Lastbewertung ist zeitgesteuert, wenn der Regelkreis in bestimmten Zeitintervallen durchlaufen wird. □

In einem überlasteten System verursachen überlastgesteuerte Verfahren ein ständiges Durchlaufen des Regelkreises, obwohl keine Möglichkeit zur Lastverteilung besteht. Analog verhält es sich bei unterlastgesteuerte Verfahren in unterlasteten Systemen. Dieses Problem tritt bei zeitgesteuerten Verfahren nicht auf. Allerdings ist hier die Bestimmung eines geeigneten Zeitintervalls schwierig.

3.2.3 Lastverteilung

Die Lastverteilung greift aktiv in ein verteiltes System ein, um die Arbeitslast zwischen den einzelnen Ressourcen aufzuteilen. Um eine klare Terminologie für diesen Bereich der Lastverwaltung zu schaffen, muß zuerst zwischen Lastverteilungseinheiten und Ausführungseinheiten unterschieden werden.

Definition 3.17 *Lastverteilungseinheiten und Ausführungseinheiten*

Als Lastverteilungseinheiten bezeichnet man die Komponenten eines verteilten Systems, die der Lastverteilung unterliegen.

Ausführungseinheiten sind die Ressourcen eines verteilten Systems, welche Leistung zur Verfügung stellen, damit die Lastverteilungseinheiten abgearbeitet werden können. □

Lastverteilungseinheiten konsumieren bei ihrer Abarbeitung Leistung und erzeugen dadurch Last. Die Lastverteilung weist ihnen, gemäß den Anweisungen der Lastbe-

wertung, Ausführungseinheiten zu, welche die benötigte Leistung zur Verfügung stellen. Dadurch wird indirekt die Verteilung der Last auf die einzelnen Ausführungseinheiten gesteuert. Typische Kombinationen von Ausführungs- und Lastverteilungseinheiten sind Rechner, die Prozesse ausführen, Prozesse, die Berechnungen über Daten durchführen, und Objekte, die Anfragen abarbeiten. Wie am Beispiel der Prozesse zu sehen ist, kann eine Einheit je nach Betrachtungsweise, entweder Ausführungs- oder Lastverteilungseinheit sein.

Die Wahl der Lastverteilungseinheiten bestimmt die Granularität der Lastverteilung. Bei vielen Ausführungseinheiten, wie beispielsweise dem Prozessor, hängt die Leistungsaufnahme der Verbraucher von der Zeitdauer der Ressourcennutzung ab. Eine Lastverteilungseinheit benötigt dann um so mehr Leistung, je länger sie die ihr zugewiesene Ausführungseinheit beansprucht. Dadurch vergrößert sich die Zeitspanne bis die Last neu verteilt werden kann, was wiederum die Granularität der Lastverteilung vergröbert. Je grober die Granularität ist, desto schlechter ist auch die Qualität der Lastverwaltung, da Lastausgleich und Lastbalancierung schwerer zu erreichen sind. Dieses Problem kann dadurch gelöst werden, daß die Abarbeitung der Lastverteilungseinheiten unterbrochen wird, um die Last neu zu verteilten.

Definition 3.18 *Unterbrechbarkeit der Lastverteilungseinheiten*

> Lastverteilungseinheiten, deren Abarbeitung zum Zweck der Lastverteilung unterbrochen werden kann, bezeichnet man als präemptiv.
>
> Ist eine Unterbrechung nicht möglich, so spricht man von nichtpräemptiven Lastverteilungseinheiten.
>
> Als bedingt-präemptiv bezeichnet man Lastverteilungseinheiten, deren Abarbeitung nur an bestimmten Punkten unterbrechbar ist. □

Präemptive Lastverteilungseinheiten ermöglichen es, die Granularität der Lastverteilung zu verfeinern und somit die Qualität der Lastverwaltung zu erhöhen.

Ein weiteres Klassifikationsmerkmal sind die Mechanismen der Lastverteilung. Dabei wird zwischen der Lastzuweisung und der Lastverschiebung unterschieden.

Definition 3.19 *Mechanismen der Lastverteilung*

> Von Lastzuweisung spricht man, wenn einer Ausführungseinheit Arbeitslast zugewiesen wird, indem man ihr eine Lastverteilungseinheit zuteilt.
>
> Die Verschiebung von Arbeitslast zwischen mehreren Ausführungseinheiten durch Umverteilung von Lastverteilungseinheiten bezeichnet man als Lastverschiebung. □

Ein Verfahren zur Lastzuweisung ist die Initialplazierung [65]. Dabei wird einer neu erzeugten Lastverteilungseinheit erstmals eine Ausführungseinheit zugewiesen. Die Initialplazierung ist ein sehr wirkungsvolles und gleichzeitig kostengünstiges Verfahren, da sie kaum Mehraufwand verursacht. Im Gegensatz zur Lastzuweisung sind für die Lastverschiebung präemptive oder zumindest bedingt-präemptive Lastverteilungseinheiten nötig, da die Umverteilung von Arbeitslast die Unterbrechung bestehender

Ausführungseinheiten erfordert. Als konkrete Verfahren zur Lastverschiebung unterscheidet man zwischen der Migration und der Replikation:

- Die Migration bezeichnet die Verschiebung einer bestehenden Lastverteilungseinheit zu einer neuen Ausführungseinheit. Dadurch verringert sich die Last der ursprünglichen und die Last der neuen Ausführungseinheit steigt.
- Bei der Replikation wird eine Lastverteilungseinheit auf mehrere Ausführungseinheiten verteilt, wobei sogenannte Replikate entstehen. Das Verfahren muß dabei gewährleisten, daß sich die Leistungsaufnahme der ursprünglichen Lastverteilungseinheit auf alle Replikate verteilt. Dadurch sinkt die Last der ursprünglichen Ausführungseinheit und die Last der neu hinzugenommenen erhöht sich.

Der Nutzen der Migration wurde anfangs kontrovers diskutiert. LELAND und OTT [65] kamen zu dem Schluß, daß die Migration, neben der Initialplazierung, wesentliche Vorteile für die Lastverwaltung bringt. EGAR u.a. [26] wiederum folgerten aus der Untersuchung von Warteschlangenmodellen, daß keine deutlichen Verbesserungen erzielt werden können. Die Arbeit von LITZKOW und LIVNY [68] zeigte schließlich, daß die Migration von langlebigen Prozessen durchaus sinnvoll ist. Mittlerweile gibt es eine Vielzahl von Untersuchungen die sich insbesondere mit der Prozeßmigration beschäftigen [138, 129]. Diese Arbeiten bestätigen, daß die Migration dann zweckmäßig ist, wenn die Lastverteilungseinheiten langlebig genug sind, so daß der Mehraufwand, der durch die Migration entsteht, von der erzielten Leistungssteigerung aufgewogen wird [43, 15, 85].

Eines der Haupteinsatzgebiete der Replikation ist die Fehlertoleranz [41]. Sie wird eingesetzt, um die Ausfallsicherheit durch redundante Einheiten (Replikate) zu erhöhen [83]. Als Lastverteilungsmechanismus ermöglicht es die Replikation, Überlast abzubauen, indem die Last auf mehrere Replikate verteilt wird. Im Gegensatz zur Migration kommt die Replikation im Bereich der Lastverwaltung nur selten zum Einsatz und beschränkt sich dabei meist auf zustandslose Lastverteilungseinheiten [102, 100]. Bei der Verwendung von zustandsbehafteten Einheiten muß die Konsistenz des Zustands der Replikate durch spezielle Protokolle gewährleistet werden. Die Konsistenzprotokolle aus dem Bereich der Fehlertoleranz, wie z.B. Gruppenkommunikation [9, 112, 80] und Two-Phase Commit [74], sind im Bereich der Lastverwaltung unbrauchbar, da sie statt einer Verteilung eine Vervielfachung der Arbeitslast bewirken. Im Bereich der Lastverteilung wurden effiziente Konsistenzprotokolle bisher nur auf der Ebene der Anwendung realisiert [76, 127]. Die Replikation als Mechanismus der Lastverschiebung wird in Kapitel 4.3 ausführlich behandelt.

Wie schon die Lasterfassung, kann auch die Lastverteilung auf unterschiedlichen Ebenen realisiert werden.

Definition 3.20 *Integration der Lastverteilung*

Die Lastverteilung kann auf verschiedenen Ebenen eines verteilten Systems realisiert werden. Sie kann in das Rechensystem, das Ablaufsystem und die Anwendungen integriert sein. □

In der Regel ist die Lastverteilung auf der Ebene der Anwendung oder des Ablaufsystems realisiert. Die Integration in das Rechensystem erfordert entsprechend angepaßte Hardware-Komponenten, die nur in bestimmten Teilbereichen wie z.B. bei Netzwerk-Routern eingesetzt werden [18].

Als letztes Klassifikationsmerkmal dieses Teilbereichs wird nun die Transparenz der Lastverteilung betrachtet:

Definition 3.21 *Transparenz der Lastverteilung*

> Die Lastverteilung ist transparent, wenn dadurch für den Entwickler einer verteilten Anwendung kein Mehraufwand entsteht. Anderenfalls ist sie nicht-transparent. □

Die Transparenz wird durch den Mehraufwand des Entwicklers bestimmt, der durch die Lastverteilung entsteht. Sie hängt teilweise von der Integration der Lastverteilung ab. Anwendungsintegrierte Lastverteilung ist nicht-transparent, da sie stets vom Entwickler realisiert werden muß. Ist die Lastverteilung in das Ablaufsystem oder das Rechensystem integriert, so ist sie in der Regel transparent, da kein Mehraufwand für den Entwickler entsteht.

3.2.4 Die Klassifikation im Überblick

Die in den vorangegangenen Abschnitten aufgestellte Klassifikation der Lastverwaltung ist in Tabelle 3.1 im Überblick dargestellt. Die Klassifikationsmerkmale sind in die Abschnitte Lasterfassung, Lastbewertung und Lastverteilung gegliedert. Dies ermöglicht es, die Komponenten der Lastverwaltung getrennt voneinander zu betrachten, was die Klassifikation erheblich erleichtert. Für die einzelnen Komponenten sind die entsprechenden Kriterien und die jeweils möglichen Varianten aufgezählt. Bei der Betrachtung konkreter Systeme erscheint nur eine Teilmenge der hier aufgelisteten Kriterien, da nicht alle Ausprägungen eines Klassifikationskriteriums mit allen anderen vereinbar sind. Die Abhängigkeiten zwischen den einzelnen Klassifikationsmerkmalen wurden bereits in den vorangegangenen Abschnitten beschrieben.

3.3 Einige ausgewählte Beispiele

In diesem Abschnitt werden einige ausgewählte Ansätze zur Lastverwaltung näher beschrieben und in das vorgestellte Klassifikationsschema eingeordnet. Die Anzahl der Systeme, die einer so detaillierten Klassifikation unterzogen werden können, ist trotz der sehr großen Zahl an Lastverwaltungssystemen begrenzt. Dies liegt daran, daß die Dokumentation vieler Systeme in einigen Teilbereichen unzureichend ist, was eine vollständige Klassifikation unmöglich macht. Die Klassifikation einiger ausgewählter Lastverwaltungssysteme soll sowohl die einfache und allgemeine Anwendbarkeit des Klassifikationsmodells demonstrieren als auch die Kompaktheit der daraus re-

Tabelle 3.1 Klassifikationsmerkmale der Lastverwaltung

Komponente	Klassifikationskriterium	Varianten
Lasterfassung	Ebenen	Rechensystem, Ablaufsystem, Anwendung
	Ressourcen	Prozessor, Speicher, Netzwerk, Prozesse, Threads, Server, Dienste, Objekte u.a.
	Integration	Rechensystem, Ablaufsystem, Anwendung
	Transparenz	transparent, nicht−transparent
Lastbewertung	Ziel	Lastverteilung, Lastbalancierung
	Lokalisation	lokal, gruppiert, zentral
	Interaktion der Last−bewertungskomponenten	kooperierend, autonom
	Systemkenntnis	lokal, partiell, global
	Wirkungsbereich	minimal, begrenzt, maximal
	Informationsgrundlage	statisch, dynamisch
	Adaptivität	adaptiv, nicht−adaptiv
	Steuerung und Initiierung	lastgesteuert, überlastgesteuert, unterlastgesteuert, zeitgesteuert
Lastverteilung	Lastverteilungseinheiten	Daten, Anfragen, Transaktionen, Prozesse, Threads, Server, Dienste, Objekte u.a.
	Unterbrechbarkeit der Lastverteilungseinheiten	präemptiv, bedingt−präemptiv, nicht−präemptiv
	Ausführungseinheiten	Rechner, Prozesse, Threads, Dienste, Objekte u.a.
	Mechanismen	Lastzuweisung, Lastverschiebung
	Lastzuweisung	Initialplazierung u.a.
	Lastverschiebung	Migration, Replikation u.a
	Integration	Rechensystem, Ablaufsystem, Anwendung
	Transparenz	transparent, nicht−transparent

sultierenden Beschreibungen verdeutlichen. Im einzelnen werden folgende Systeme untersucht:

- ALDY (Adaptive Load Distribution System) [119]
- LoMan (Load Management Facility) [109]
- TAO Lastverwaltung [100, 101]

Diese Systeme wurden ausgewählt, da sie sich sowohl in ihrer Zielsetzung als auch in den verwendeten Mechanismen stark unterscheiden und somit einen guten Eindruck von den vielfältigen Möglichkeiten der Lastverwaltung vermitteln.

3.3.1 ALDY

Das Lastverwaltungssystem ALDY stellt dem Entwickler eine Programmierbibliothek zur Verfügung, die als Rahmen zur Entwicklung lastverwalteter, paralleler Anwendungen dient. ALDY ist nicht an ein spezielles Programmiermodell gebunden und erlaubt es dem Entwickler, die Lastverteilungseinheiten selbst zu bestimmen. Die Klassifikation ist in Tabelle 3.2 dargestellt.

Als Ausführungseinheiten werden virtuelle Prozesse verwendet, die in der Regel identisch mit Betriebssystemprozessen sind. Lastverteilungseinheiten sind sogenannte virtuelle Agenten, die eine sequentielle Folge von Aufgaben abarbeiten. Die Beschaffenheit der virtuellen Agenten ist, wie bereits erwähnt, nicht fest vorgegeben, sondern wird vom Entwickler spezifiziert. Ein virtueller Prozeß kann einen oder mehrere Agenten ausführen. Die Lastverteilungseinheiten sind bedingt-präemptiv, da sie nur zwischen zwei aufeinanderfolgenden Aufgaben unterbrechbar sind. Als Mittel zur Lastverteilung bietet ALDY die Migration von Agenten auf Prozesse an. Die Funktionalität zur Lastverschiebung muß vom Entwickler bereitgestellt werden. Es handelt sich also um nicht-transparente, anwendungsintegrierte Lastverteilung.

Ressourcen der Lasterfassung sind virtuelle Prozesse, deren Auslastung durch die Anzahl der aktiven Agenten beschrieben wird. Dazu muß der Entwickler die aktiven Bereiche der virtuellen Agenten kennzeichnen, weshalb man die Lasterfassung als nicht-transparent bezeichnen kann. Die Auswertung der Lastinformation übernimmt das Ablaufsystem. Deshalb ist die Lasterfassung sowohl in die Anwendung als auch das Ablaufsystem integriert.

Die Lastbewertung verfolgt das Ziel der Lastbalancierung. Jeder virtuelle Prozeß beinhaltet eine Lastbewertungskomponente, die zur Entscheidungsfindung Lastinformation über Prozesse heranzieht, die benachbarte[3] Agenten ausführen. Die Lastbewertungskomponenten sind also lokal angeordnet, kooperierend und ihre Systemkenntnis ist partiell. Der Wirkungsbereich der Lastbewertung ist begrenzt, da Agenten nur auf Prozesse migriert werden, die benachbarte Agenten ausführen. Die Lastbewertung ist adaptiv und dynamisch, da sie Lastinformation über Prozesse verwendet um sich an

[3] ALDY erlaubt es, Nachbarschaftsbeziehungen zwischen Agenten zu definieren, um damit deren intensive Kommunikation zu kennzeichnen.

Tabelle 3.2 Klassifikationsmerkmale des Lastverwaltungssystems ALDY

Komponente	Klassifikationskriterium	Varianten
Lasterfassung	Ebenen	Anwendung
	Ressourcen	virtuelle Prozesse
	Integration	Ablaufsystem und Anwendung
	Transparenz	nicht−transparent
Lastbewertung	Ziel	Lastbalancierung
	Lokalisation	lokal
	Interaktion der Last−bewertungskomponenten	kooperierend
	Systemkenntnis	partiell
	Wirkungsbereich	begrenzt
	Informationsgrundlage	dynamisch
	Adaptivität	adaptiv
	Steuerung und Initiierung	unterlastgesteuert
Lastverteilung	Lastverteilungseinheiten	virtuelle Agenten
	Unterbrechbarkeit der Lastverteilungseinheiten	bedingt−präemptiv
	Ausführungseinheiten	virtuelle Prozesse
	Mechanismen	Lastverschiebung
	Lastverschiebung	Migration
	Integration	Anwendung
	Transparenz	nicht−transparent

eine veränderte Lastsituation anzupassen. Hintergrundlast wird dabei implizit berücksichtigt, da sie die Last der virtuellen Prozesse erhöht. Die Initiierung der Lastbewertung erfolgt durch ein unterlastgesteuertes Verfahren.

Der wesentliche Vorteil von ALDY liegt in der freien Bestimmbarkeit der Lastverteilungseinheiten und damit auch der Granularität der Lastverteilung. Dies ist jedoch für den Programmierer mit erheblichem Mehraufwand verbunden, da die Lastverteilungseinheiten in der Regel keine natürlichen Verteilungseinheiten wie Prozesse oder Objekte sind, sondern Datenstrukturen, die eng mit der Anwendung verwoben sind. Die Identifikation und das Herauslösen dieser Datenstrukturen kann, zumindest für bestehende Anwendungen, ein unlösbares Problem darstellen [118].

3.3.2 LoMan

LoMan ist ein Lastverwaltungssystem, das für den Einsatz in Message-Passing-Umgebungen konzipiert wurde. Herausragende Merkmale von LoMan sind die Un-

Tabelle 3.3 Klassifikationsmerkmale des Lastverwaltungssystems LoMan

Komponente	Klassifikationskriterium	Varianten
Lasterfassung	Ebenen	Rechensystem, Ablaufsystem und Anwendung
	Ressourcen	Prozessor, Speicher, Netzwerk, Prozesse u.a.
	Integration	Ablaufsystem
	Transparenz	transparent
Lastbewertung	Ziel	Lastbalancierung
	Lokalisation	zentral
	Systemkenntnis	global
	Wirkungsbereich	maximal
	Informationsgrundlage	dynamisch
	Adaptivität	adaptiv
	Steuerung und Initiierung	zeitgesteuert
Lastverteilung	Lastverteilungseinheiten	Prozesse
	Unterbrechbarkeit der Lastverteilungseinheiten	präemptiv
	Ausführungseinheiten	Rechner
	Mechanismen	Lastzuweisung und Lastverschiebung
	Lastzuweisung	Initialplazierung
	Lastverschiebung	Migration
	Integration	Ablaufsystem
	Transparenz	transparent

terstützung heterogener Rechensysteme und Verwendung der präemptiven Prozeßmigration. Die Klassifikation ist in Tabelle 3.3 dargestellt.

Die Lastverteilung beherrscht sowohl die Initialplazierung als auch die Migration von Prozessen. Ein Mechanismus zur Prozeßmigration in einem verteilten System muß gewährleisten, daß keine inkonsistenten Zustände bei den beteiligten Prozessen entstehen. In nachrichtenorientierten Systemen kommunizieren Prozesse über Nachrichtenaustausch. Der Verlust oder die Vervielfältigung von Nachrichten während der Migration würde zu inkonsistenten Zuständen bei den beteiligten Prozessen führen. Der Migrationsmechanismus muß also ein Synchronisationsprotokoll zur Verfügung stellen, das Prozesse während der Migration davon abhält, Nachrichten zu empfangen oder zu versenden. Weiterhin müssen die Kommunikationsverbindungen zwischen den beteiligten Prozessen vor der Migration getrennt und anschließend wieder aufgebaut werden. Zu diesem Zweck verwendet LoMan das Werkzeug CoCheck (Consistent Checkpoints). CoCheck ermöglicht die Erstellung konsistenter Sicherungspunk-

te für nachrichtenorientierte Anwendungen und ist sowohl für PVM [128] als auch für MPI [129] verfügbar. CoCheck erfordert keine Änderung der Anwendung, da es ausschließlich Mechanismen des Betriebssystems und der jeweiligen Kommunikationsbibliothek verwendet. Die Lastverteilung ist also systemintegriert und transparent. Die Erstellung von Sicherungspunkten ist jederzeit möglich, d.h. die Migration ist präemptiv.

LoMan verwendet zur Lasterfassung den Node Status Reporter (NSR) [110]. Der NSR ist ein Werkzeug, das Lastinformation auf allen Ebenen eines verteilten Systems mißt. Als Ressourcen des Rechensystems werden der Prozessor, der Speicher und das Netzwerk betrachtet. Auf der Ebene des Ablaufsystems findet man Information über lokale und entfernte Benutzer. Die Lastinformation über Prozesse ist schließlich auf der Ebene der Anwendung angesiedelt. Zur Erfassung der Meßdaten greift der NSR ausschließlich auf das Betriebssystem zurück, d.h die Lasterfassung ist in das Ablaufsystem integriert und deswegen auch transparent. Eine Besonderheit des NSR ist die Unterstützung heterogener Rechensysteme. Um die Lastwerte heterogener Ressourcen vergleichbar zu machen, werden diese bezüglich eines Einheitsmaßes normiert. Die Einheitsmaße für die unterschiedlichen Ressourcen ermittelt der NSR durch Leistungsvergleiche (Benchmarks).

Die Lastbewertung ist eine zentrale Komponente und verfügt folglich über globale Systemkenntnis und hat einen maximalen Wirkungsbereich. Das Ziel der Lastbewertung ist die Lastbalancierung im verteilten System, wobei jedoch nur Rechner genutzt werden, auf welchen kein lokaler Benutzer arbeitet. Das Verfahren ist adaptiv, da sowohl Hintergrundlast als auch lokale Benutzer berücksichtigt werden. Die Lastbewertung ist zeitgesteuert, was sich bei langlaufenden Prozessen anbietet, da somit unnötig häufige Durchläufe des Regelkreises bei stark unter- bzw. überlasteten Systemen vermieden werden. Ein wesentliches Merkmal von LoMan ist die Kostensensitivität. Die Lastbewertung berücksichtigt bei der Entscheidungsfindung neben der Lastsituation auch die erwarteten Kosten der Lastverteilung. Dadurch können Aktionen vermieden werden, deren voraussichtliche Kosten den erwarteten Nutzen übersteigen.

LoMan zeichnet sich durch seine Lasterfassung aus, die vielfältige Ressourcen auf allen Ebenen eines verteilten Systems betrachtet. Die detaillierte Lastinformation und die Kostensensitivität der Lastbewertung ermöglichen eine qualitativ hochwertige Lastbewertung. Die präemptive Prozeßmigration gewährleistet schließlich, daß die Entscheidungen der Lastbewertung jederzeit schnell umgesetzt werden können.

3.3.3 TAO Lastverwaltung

Die Lastverwaltung der CORBA Implementierung TAO [115] ist in Anlehnung an die CORBA Spezifikation zur Fehlertoleranz [92] entstanden. Die Parallelen sind insbesondere in der Anwendung und der Realisierung der Objektreplikation zu sehen. Tabelle 3.4 zeigt die Klassifikation der TAO Lastverwaltung.

Die Lastverteilung basiert auf der Replikation zustandsloser Objekte, wodurch auf Konsistenzprotokolle, wie sie in Abschnitt 3.2.3 beschrieben sind, verzichtet werden kann. Als Ausführungseinheiten dienen Server-Prozesse bzw. die Server-Rechner,

Tabelle 3.4 Klassifikationsmerkmale des Lastverwaltungssystems TAO

Komponente	Klassifikationskriterium	Varianten
Lasterfassung	Ebenen	Anwendung
	Integration	Anwendung
	Transparenz	nicht−transparent
Lastbewertung	Ziel	Lastverteilung und Lastbalancierung
	Lokalisation	zentral
	Systemkenntnis	global
	Wirkungsbereich	maximal
	Informationsgrundlage	statisch und dynamisch
	Adaptivität	adaptiv und nicht−adaptiv
Lastverteilung	Lastverteilungseinheiten	Objekte
	Unterbrechbarkeit der Lastverteilungseinheiten	bedingt−präemptiv
	Ausführungseinheiten	Server−Prozesse und Server−Rechner
	Mechanismen	Lastverschiebung
	Lastverschiebung	Replikation
	Integration	Ablaufsystem und Anwendung
	Transparenz	nicht−transparent

welche die Prozesse ausführen. Die Replikation ist bedingt-präemptiv, da CORBA Objekte nur zwischen aufeinanderfolgenden Anfragen unterbrochen werden. Die Funktionalität zur Replikation stellt der Entwickler bereit, indem er vorgegebene Schnittstellen des Lastverwaltungssystems implementiert. Das Ablaufsystem initiiert und steuert die Replikation und gewährleistet, daß keine Anfragen verlorengehen oder vervielfältigt werden. Folglich ist die Lastverteilung sowohl in das Ablaufsystem als auch in die Anwendung integriert und damit nicht-transparent.

Analog zur Lastverteilung muß auch die Funktionalität zur Lasterfassung vom Entwickler über Schnittstellen des Lastverwaltungssystems zur Verfügung gestellt werden. Die Lasterfassung ist dadurch anwendungsintegriert und nicht-transparent. Die Art und den Umfang der bereitgestellten Lastinformation kann der Entwickler frei wählen. Sie ist jedoch wegen der Integration in die Anwendung größtenteils auf die Anwendungsebene beschränkt.

Die Lastbewertung der TAO Lastverwaltung ist frei konfigurierbar und damit sehr flexibel. Es stehen sowohl statische und nicht-adaptive als auch dynamische und adaptive Verfahren zur Auswahl. Die Lastbewertung ist als zentrale Komponente mit globaler Systemkenntnis und maximalem Wirkungsbereich realisiert.

Die Anlehnung der Lastverwaltung an die Spezifikation zur Fehlertoleranz erleichtert zukünftige Bestrebungen hinsichtlich der Integration und der Interoperabilität der bei-

den Dienste. Wesentliche Nachteile der TAO Lastverwaltung sind in der mangelnden Transparenz und der fehlenden Unterstützung der Migration zu sehen. Die zusätzliche Beschränkung der Replikation auf zustandslose Objekte engt den Spielraum der Lastverteilung weiter ein. Damit ist der Einsatz der Lastverwaltung auf eine kleine Auswahl von Anwendungen beschränkt.

3.4 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie man ein Lastverwaltungssystem mit Hilfe der Terminologie der Regelungstechnik in logische Komponenten zerlegen kann. Darauf folgte eine Beschreibung der einzelnen Komponenten und ihre Klassifikation anhand zahlreicher Merkmale. Zum Schluß des Kapitels wurden drei konkrete Lastverwaltungssysteme ausgewählt um die Anwendung des vorgestellten Klassifikationsmodells zu demonstrieren und einen Einblick in die Möglichkeiten der Lastverwaltung zu geben.

Die Aufteilung der Lastverwaltung in ihre einzelnen Komponenten, die Lasterfassung, die Lastbewertung und die Lastverteilung, schafft wesentliche Vorteile für die Klassifikation. Das vorgestellte Klassifikationsschema gewährleistet, daß alle Komponenten gleichermaßen berücksichtigt werden. Das hilft wiederum Lücken und Inkonsistenzen in der Terminologie zu vermeiden, da der Zwang zu einer ganzheitlichen Betrachtung der Lastverwaltung besteht. Trotzdem bleibt die Klassifikation wegen der Trennung der einzelnen Komponenten übersichtlich und handhabbar.

Der Schwerpunkt des Klassifikationsmodells liegt auf der Beschreibung der grundlegenden Eigenschaften und der Funktionsweise von Lastverwaltungssystemen. Konkrete Strategien und Algorithmen zur Lastbewertung sind in dem Modell nicht enthalten, da diese eine wesentlich detailliertere Betrachtungsweise voraussetzen. Schließlich soll die Klassifikation einen Überblick bieten und somit die Grundlage für den Vergleich unterschiedlicher Systeme schaffen. Für eine detaillierte Beschreibung von Strategien und Algorithmen zur Lastbewertung wird in Kapitel 5 der Begriff des Lastmodells eingeführt. Das Klassifikationsmodell dient aber nicht nur der Darstellung bestehender Lastverwaltungssysteme. Wie in Kapitel 4 gezeigt wird, ist es auch geeignet, um neue Lastverwaltungssysteme zu planen und zu konzipieren.

4. Lastverwaltung in verteilten objektorientierten Systemen

In den vorangegangenen Kapiteln wurden verteilte Systeme und deren Programmiermodelle vorgestellt. Die Programmiermodelle haben sich im Lauf der Zeit stets an die wachsenden Anforderungen der Anwendungen angepaßt. Die Entwicklung reicht von nachrichtenorientierten Systemen, über den entfernten Prozeduraufruf, bis hin zu verteilten objektorientierten Systemen. Ein weiterer Schwerpunkt war die Lastverwaltung für verteilte Systeme. Ausgehend von der Komponentenstruktur der Lastverwaltung wurde ein Klassifikationsschema erarbeitet, das es ermöglicht, die Eigenschaften und die Funktionsweise von Lastverwaltungssystemen zu beschreiben und zu vergleichen.

Ziel dieses Kapitels ist es, ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen zu entwickeln. Dazu müssen zunächst die Besonderheiten verteilter objektorientierter Systeme betrachtet werden, um daraus die, für dieses Programmiermodell spezifischen Anforderungen an die Lastverwaltung ableiten zu können. Mit Hilfe des Klassifikationsschemas aus dem vorangegangenen Kapitel wird dann aus diesen Anforderungen ein neues Lastverwaltungskonzept entwickelt. Anschließend werden einige Besonderheiten, wie die Migration und die Replikation von Objekten, näher erläutert. Das Kapitel schließt mit einem Vergleich des hier vorgestellten Konzepts mit bestehenden Lastverwaltungssystemen, um die Notwendigkeit für eine Neuentwicklung aufzuzeigen.

4.1 Merkmale verteilter objektorientierter Systeme

Verteilte objektorientierte Systeme zeichnen sich durch eine Reihe von Merkmalen aus, die sie von anderen Programmiermodellen abgrenzen. Diese Merkmale können dazu benutzt werden, um die spezifischen Anforderungen der Lastverwaltung in verteilten objektorientierten Systemen zu identifizieren und daraus ein geeignetes Konzept zur Lastverwaltung abzuleiten. Im einzelnen handelt es sich um folgende Charakteristika:

- Verteiltheit: Wie bereits erwähnt, gliedern sich verteilte objektorientierte Anwendungen in Clients und Server. Server sind Subsysteme, die Objekten als

Laufzeitumgebung dienen. Meist sind Server mit Prozessen gleichzusetzen, die von sogenannten Server-Rechnern ausgeführt werden. Objekte bieten Dienste in Form von Methoden an und stellen die eigentlichen Verteilungseinheiten dar. Clients laufen auf sogenannten Client-Rechnern und fragen die Methoden verteilter Objekte über den Mechanismus des entfernten Methodenaufrufs an. Verteilte objektorientierte Anwendungen können sowohl eng als auch lose gekoppelt sein. Eng gekoppelte Anwendungen sind in der Regel auf ein lokales Netz beschränkt und zeichnen sich somit durch performante Kommunikation aus. Im Gegensatz dazu sind lose gekoppelte Anwendungen durch ein Weitverkehrsnetz verbunden, was die Performanz der Kommunikation erheblich verringert. Die örtliche Ausdehnung eines verteilten objektorientierten Systems kann also von einem lokalen Netz bis hin zu einem weltweit verteilten System reichen. Aufgrund der Globalisierung der Geschäftsprozesse haben weltweit verteilte Systeme in letzter Zeit stark an Bedeutung gewonnen.

- Heterogenität: Verteilte objektorientierte Systeme sind in zweierlei Hinsicht heterogen. Einerseits können in einem verteilten System unterschiedliche Rechner und Betriebssysteme existieren, was dazu führt, daß Anwendungen hinsichtlich der nutzbaren Ressourcen eingeschränkt werden. Andererseits können auch Anwendungen bzw. ihre Objekte heterogen bezüglich der verwendeten Programmiersprache sein. So ist es möglich, daß verschiedene Objekte einer Anwendung in unterschiedlichen Programmiersprachen realisiert sind. Die Heterogenität hat entscheidenden Einfluß auf die Qualität der Lastverwaltung, da sie die Möglichkeiten der Lastverteilung und insbesondere der Lastverschiebung stark einschränkt [109].

- Transparenz: Programmiermodelle für verteilte objektorientierte Systeme basieren auf spezifischen Objektmodellen, welche die Beschaffenheit der Objekte und den Ablauf der Kommunikation beschreiben [98, 24, 136]. Diese Objektmodelle stellen implizit Anforderungen hinsichtlich der Transparenz der verteilten Objekte: Die Ortstransparenz fordert, daß der Ausführungsort eines Objekts seinen Clients nicht bekannt sein darf. Dadurch wird die Verteiltheit der Anwendung vor dem Benutzer verborgen. Die Zugriffstransparenz legt fest, daß alle Objekte auf dieselbe Art und Weise angefragt werden, unabhängig vom Rechensystem, der verwendeten Programmiersprache und der Implementierung. Dies gewährleistet homogene Kommunikation zwischen den Verteilungseinheiten, was dazu führt, daß die Heterogenität des verteilten Systems verborgen bleibt. Die Lastverwaltung sollte sich möglichst nahtlos in das vorgegebene Objektmodell integrieren, um den Mehraufwand für den Benutzer so gering als möglich zu halten. Dazu muß gewährleistet sein, daß die Lastverwaltung den Transparenzforderungen des zugrundeliegenden Objektmodells genügt.

- Granularität: Verteilte objektorientierte Anwendungen sind wesentlich feingranularer als nachrichtenorientierte Systeme, da neben Prozessen auch Objekte und Anfragen als Lastverteilungseinheiten in Frage kommen. Dies eröffnet neue Möglichkeiten für die Lastverwaltung, da die Lastverteilung wesentlich fein-

granularer erfolgen kann, was wiederum die Wirksamkeit der Lastverwaltung steigert. Andererseits erhöht sich dadurch die Komplexität der Lastverteilungsmechanismen, da keine Standardwerkzeuge, wie beispielsweise der in Kapitel 3.3.2 beschriebene CoCheck, verwendet werden können.

- Offenheit: Die Kommunikationsstruktur verteilter objektorientierter Anwendungen ist offen hinsichtlich der Nutzung der bereitgestellten Dienste. Clients verwenden Objektreferenzen, um einzelne Objekte zu adressieren und deren Dienste zu nutzen. Objektreferenzen können auch an andere Clients weitergegeben werden, was dazu führt, daß die Anzahl der Clients eines Objekts potentiell unbegrenzt ist. Wegen der Offenheit verteilter objektorientierter Systeme können Objekte und somit auch die ausführenden Rechner aufgrund einer zu großen Anzahl von Anfragen überlastet werden. Neben der in Kapitel 3.2.2 erwähnten Hintergrundlast ist die sogenannte Anfragelast eine weitere Ursache für die Entstehung von Überlast. Ein Lastverwaltungssystem muß geeignete Lastverteilungsmechanismen bereitstellen, um Hintergrundlast und Anfragelast entgegenwirken zu können.

Insbesondere die Transparenz, die Granularität und die Offenheit sind Merkmale, die verteilte objektorientierte Systeme von anderen Programmiermodellen abgrenzen. Diese Charakteristika stellen neue Anforderungen an die Lastverwaltung, eröffnen aber auch neue Wege und Möglichkeiten.

4.2 Entwicklung eines geeigneten Konzepts

Anhand der beschriebenen Eigenschaften verteilter objektorientierter Systeme wird nun ein geeignetes Konzept zur Lastverwaltung entwickelt.

4.2.1 Einheiten der Lastverteilung

Eine wesentliche Neuerung in verteilten objektorientierten Systemen ist die große Auswahl an möglichen Lastverteilungseinheiten. Die Wahl der Lastverteilungseinheiten bestimmt die Granularität der Lastverteilung. Je feingranularer die Lastverteilung ist, desto höher ist auch die Qualität der Lastverwaltung, da Lastausgleich und Lastbalancierung einfacher zu erreichen sind. Es stehen folgende Lastverteilungseinheiten zur Auswahl:

- Server-Prozesse: Prozesse sind natürliche Verteilungseinheiten, d.h. eine explizite Kennzeichnung der Lastverteilungseinheiten durch den Entwickler ist nicht nötig. Zur Lastverschiebung können Werkzeuge zur Erstellung von Sicherungspunkten, wie beispielsweise CoCheck, verwendet werden. Damit ist es möglich die Prozeßverschiebung transparent, d.h. ohne Eingreifen des Entwicklers, zu realisieren.

- Daten: Im Gegensatz zu Prozessen sind Daten keine natürlichen Verteilungseinheiten. Der Entwickler muß geeignete Datenstrukturen identifizieren und die Anwendung entsprechend der gewählten Lastverteilungseinheiten strukturieren. Wie bereits in Kapitel 3.3.1 gezeigt wurde, kann dies einen erheblichen Mehraufwand verursachen. Einerseits hat die Verteilung von Daten den Nachteil, daß sie, wegen der nötigen Interaktion mit dem Entwickler, nicht transparent realisierbar ist. Andererseits kann der Entwickler durch die Wahl geeigneter Datenstrukturen die Granularität der Lastverteilung frei bestimmen, wodurch die Qualität der Lastverwaltung verbessert werden kann.

- Objekte: Objekte sind natürliche Verteilungseinheiten verteilter objektorientierter Systeme. Sie sind feingranularer als Server-Prozesse, da ein Prozeß mehrere Objekte beinhalten kann. Durch die Replikation von Objekten und die Verteilung von Anfragen auf die einzelnen Replikate kann die Granularität der Lastverteilung zusätzlich verfeinert werden. Analog zur Prozeßmigration muß auch bei der Migration und der Replikation von Objekten deren Zustand berücksichtigt werden. Unter der Voraussetzung, daß geeignete Mechanismen zur Gewährleistung der Objektpersistenz vorhanden sind, kann die Lastverteilung bei Objekten transparent erfolgen. Viele verteilte objektorientierte Systeme bieten solche Persistenzmechanismen an. So kann beispielsweise in Java RMI mittels der Objektserialisierung [135] oder in CORBA mit Hilfe des Persistent State Service [93] der Zustand von Objekten dauerhaft gespeichert werden, um zu einem späteren Zeitpunkt wieder Objekte daraus zu erzeugen.

Prozesse als Lastverteilungseinheiten ermöglichen transparente Lastverteilung, sie sind jedoch relativ grobgranular. Die Verwendung von Daten erlaubt es dem Entwickler, die Granularität der Lastverteilung frei zu bestimmen, die Lastverteilung ist aber nicht transparent. Objekte als Lastverteilungseinheiten ermöglichen sowohl transparente als auch feingranulare Lastverteilung. Deshalb eignen sich Objekte am besten als Lastverteilungseinheiten für verteilte objektorientierte Systeme.

Diese Entscheidung hat weitreichende Auswirkungen auf das gesamte Lastverwaltungskonzept: Die Ausführungseinheiten sind Server-Prozesse, da sie Objekten als Laufzeitumgebung dienen. Server-Prozesse werden wiederum von Server-Rechnern ausgeführt. Wegen dieser Schichtung spricht man auch von primären und sekundären Ausführungseinheiten. Die Lasterfassung muß sowohl das Rechensystem als auch die Anwendung betrachten, um Information über Objekte als Verteilungseinheiten, Server-Prozesse als primäre und Server-Rechner als sekundäre Ausführungseinheiten zu gewinnen. Objekte sind bedingt-präemptiv, da sie nur an bestimmten Punkten ihrer Laufzeit unterbrochen werden können. Dies liegt daran, daß gängige Persistenzmechanismen die Objekte unabhängig von ihren Server-Prozessen betrachten. Server-Prozesse nehmen Anfragen vom Broker entgegen und reichen diese an die entsprechenden Objekte weiter. Dabei entscheidet der Server-Prozeß, ob Anfragen nebenläufig oder sequentiell abgearbeitet werden, indem er Anfragen sofort weiterreicht oder sie zurückhält bis die vorangegangene Anfrage abgearbeitet ist. Information über aktive und anstehende Anfragen ist also im Server-Prozeß enthalten. Da gängige Persistenzmechanismen lediglich Objekte betrachten, ginge diese Information verloren.

Folglich darf ein Objekt nur dann unterbrochen werden, wenn beim Server weder aktive noch unbearbeitete Anfragen für das Objekt vorliegen.

4.2.2 Hintergrundlast und Anfragelast

In verteilten objektorientierten Systemen gibt es zwei Ursachen für die Entstehung von Überlast: Die Hintergrundlast und die Anfragelast. Als Hintergrundlast bezeichnet man die Last, welche von Anwendungen erzeugt wird, die außerhalb der Kontrolle der Lastverwaltung liegen. Das Auftreten von Hintergrundlast kann einen Rechner bzw. seine Ressourcen überlasten. Die Lastverwaltung muß dem entgegenwirken, indem sie Arbeitslast von überlasteten Rechnern auf weniger stark belastete Rechner verschiebt.

Wegen der Offenheit verteilter objektorientierter Systeme können Server auch aufgrund einer zu großen Anzahl von Anfragen überlastet werden. Diese Art von Last bezeichnet man als Anfragelast. Im Gegensatz zur Hintergrundlast stammt sie von der lastverwalteten Anwendung selbst. Da die Anzahl der Clients eines Objekts potentiell unbegrenzt ist, kann die Anfragelast so groß werden, daß sie von keinem Rechner vollständig abgearbeitet werden kann. Eine Verschiebung des Objekts auf einen anderen Rechner schafft in diesem Fall keine Abhilfe. Deshalb muß die Lastverwaltung geeignete Mechanismen zur Verfügung stellen, um die Anfragelast eines Objekts auf mehrere Objekte bzw. deren ausführende Rechner zu verteilen.

4.2.3 Mechanismen der Lastverteilung

Die Lastverwaltung benutzt Mechanismen zur Lastverteilung, um Hintergrundlast und Anfragelast entgegenzuwirken. In verteilten objektorientierten Systemen sind folgende Mechanismen möglich:

- Initialplazierung: Die Initialplazierung beschäftigt sich mit dem Erzeugen eines neuen Objekts auf einem geeigneten Server-Rechner.
- Objektmigration: Unter Objektmigration versteht man die Verschiebung eines bestehenden Objekts von seinem ausführenden Rechner auf einen anderen Rechner.
- Objektreplikation: Bei der Replikation eines Objekts wird eine Kopie des Objekts (Replikat) auf einem neuen Rechner erzeugt. Die Anfragen an das ursprüngliche Objekt werden dann zwischen den Replikaten aufgeteilt.

Bei der Initialplazierung werden neu erzeugten Objekten geeignete Server-Rechner und somit auch Server-Prozesse zugewiesen. Hierbei handelt es sich um ein erprobtes Verfahren [65], das in vielen gängigen Lastverwaltungssystemen eingesetzt wird [119, 109, 100].

Bei der Objektmigration wird ein bestehendes Objekt von einem überlasteten Rechner auf einen anderen Rechner verschoben, der eine effizientere Ausführung verspricht.

Der Nutzen der Migration für langlaufende Lastverteilungseinheiten wird von vielen Arbeiten bestätigt [85, 68, 43]. Da es sich bei Objekten um langlaufende Einheiten handelt, ist die Objektmigration ein vielversprechendes Verfahren zur Lastverschiebung in verteilten objektorientierten Systemen. Die Objektmigration ist geeignet, um die, bei der Initialplazierung festgelegte Verteilung von Objekten auf Rechner an eine veränderte Lastsituation anzupassen. Beim Auftreten von Hintergrundlast auf einem ausführenden Rechner kann ein Objekt auf einen anderen Rechner verschoben werden, der weniger stark belastet ist. Hat ein Objekt eine zu hohe Anfragelast, so kann es mit Hilfe der Migration auf einen leistungsfähigeren Rechner verschoben werden. Die Möglichkeiten der Migration sind aber stark eingeschränkt, da die Leistungsfähigkeit eines einzelnen Rechners stets begrenzt ist. Es kann also durchaus eine Lastsituation im verteilten System auftreten, bei der jeder verfügbare Rechner durch die Ausführung eines Objekts überlastet wäre. Dies ist entweder dann der Fall, wenn alle Rechner bereits stark ausgelastet sind oder wenn die Anfragelast eines Objekts so groß ist, daß die Ausführung jeden Rechner überlasten würde.

In diesem Fall ist die Objektreplikation ein geeignetes Verfahren, um der Überlast entgegenzuwirken. Objekte werden repliziert, d.h. vervielfältigt, wobei die einzelnen Replikate auf unterschiedlichen Rechnern zur Ausführung kommen. Die Verteilung der Anfragen zwischen den einzelnen Replikaten bewirkt eine Aufteilung der Arbeitslast auf mehrere Rechner. Die Replikation verfeinert die Granularität der Lastverteilung, da sie ein Objekt in mehrere Replikate aufspaltet. Dadurch verbessert sich auch die Skalierbarkeit der verteilten Anwendung, da nun die Möglichkeit besteht, neben ganzen Objekten auch einzelne Replikate auf neu hinzugekommene Rechner umzuverteilen. Im Gegensatz zur Migration kommt die Replikation bei der Lastverteilung bisher nur selten zum Einsatz und beschränkt sich dabei meist auf zustandslose Lastverteilungseinheiten [102, 100].

In das Lastverwaltungskonzept werden die Initialplazierung, die Migration und die Replikation aufgenommen. Zu Beginn des Objekt-Lebenszyklus ermöglicht die Initialplazierung eine geeignete Verteilung von Objekten auf Rechner. Die Migration und die Replikation sind nötig, um die Entscheidung der Initialplazierung an eine veränderte Lastsituation anzupassen.

4.2.4 Transparenz der Lastverwaltung

Die Transparenz der Lastverwaltung beschreibt den Mehraufwand, der durch die Lastverwaltung für den Entwickler ensteht. Wie bereits in Abschnitt 4.1 erwähnt wurde, stellen die Objektmodelle von verteilten objektorientierten Systemen hohe Anforderungen hinsichtlich der Transparenz ihrer Programmiermodelle. Auch die Lastverwaltung sollte transparent sein, um den Mehraufwand für den Entwickler so gering als möglich zu halten und damit die Akzeptanz der Lastverwaltung zu steigern.

Transparente Lasterfassung ist nötig, da der Entwickler ansonsten mit den Ressourcen der Lasterfassung und den, von der Lastbewertung verwendeten Metriken konfrontiert wäre. Die Abhängigkeit von einer konkreten Lastbewertungskomponente würde jedoch die Portierbarkeit der Anwendungen erheblich einschränken.

Auch die Lastverteilung sollte transparent sein, um den Mehraufwand für den Entwickler so gering als möglich zu halten. Dabei muß insbesondere die Beibehaltung der Orts- und der Zugriffstransparenz berücksichtigt werden. Die Forderung der Ortstransparenz muß insofern erweitert werden, daß sie auch bei einer Migration, d.h. einer Veränderung des Ausführungsorts eines Objekts, erhalten bleibt. Dies führt zum Begriff der Migrationstransparenz:

Definition 4.1 *Migrationstransparenz*

> Die Migrationstransparenz fordert, daß jede Veränderung des Ausführungsorts eines Objekts seinen Clients verborgen bleibt. □

Die Migration hat folglich keine Auswirkung auf die Clients eines Objekts und ist damit für die Client-Seite transparent. Im Fall der Replikation muß die Zugriffstransparenz auf die Replikate eines Objekts ausgedehnt werden, was man als Replikationstransparenz bezeichnet:

Definition 4.2 *Replikationstransparenz*

> Die Replikationstransparenz fordert, daß die Clients eines Objekts keine Kenntnis darüber erlangen dürfen, daß mehrere Replikate eines Objekts existieren. □

Ein Client kann also ein repliziertes Objekt auf dieselbe Art und Weise anfragen wie ein nicht-repliziertes. Damit ist auch die Replikation für die Client-Seite transparent. Migrations- und Replikationstransparenz gewährleisten die Transparenz der Lastverteilung auf der Client-Seite. Darüber hinaus stellt transparente Lastverteilung auf der Server-Seite sicher, daß sich die Lastverwaltung nahtlos in das zugrundeliegende Programmiermodell integriert. Dies trägt letztendlich dazu bei, daß die Akzeptanz der Lastverwaltung bei den Entwicklern steigt.

4.2.5 Integration der Lastverwaltung

Die Integration der Lastverwaltung ist ausschlaggebend dafür, ob die Anforderungen hinsichtlich der Transparenz erfüllt werden können.

Eine Integration der Lastverwaltung in die Anwendung scheidet aus, da sie für den Entwickler in der Regel nicht transparent ist. Dies trifft insbesondere auf die Lastverteilung zu; aber auch die Lasterfassung auf Anwendungsebene erfordert meist ein Eingreifen des Entwicklers, außer es stehen automatische Werkzeuge mit entsprechender Funktionalität, wie z.B. ein Compiler, zur Verfügung.

Die Integration der Lasterfassung in das Rechensystem erfordert, wie in Kapitel 3.2.1 beschrieben, Hardwaremonitore, die in handelsüblichen Rechnern nur in sehr begrenztem Umfang enthalten sind. Zudem kann auf der Ebene des Rechensystem nur wenig Lastinformation über die Anwendung ermittelt werden, was jedoch eine zentrale Forderung dieses Lastverwaltungskonzepts darstellt. Da die Integration auf Anwendungsebene wegen der fehlenden Transparenz von vornherein verworfen wurde, bleibt nur

noch die Integration in das Ablaufsystem. Das Ablaufsystem ist das Bindeglied zwischen dem Rechensystem und der Anwendung. Deshalb findet man im Ablaufsystem Lastinformation über alle Ebenen eines verteilten Systems. Zudem ist die Integration auf dieser Ebene für den Entwickler transparent.

Auch bei der Integration der Lastverteilung ist das Rechensystem ungeeignet, da die verfügbaren Hardwarekomponenten wie beispielsweise Netzwerk-Router nur über sehr eingeschränkte Möglichkeiten der Lastverteilung verfügen. Deshalb muß auch die Lastverteilung in das Ablaufsystem integriert werden. Bei verteilten objektorientierten Systemen bietet sich dafür insbesondere der Broker an. Der Broker ist eine logisch zentrale Einheit des verteilten Ablaufsystems, die als Vermittler zwischen Clients und Objekten auftritt. Zu seinen Aufgaben gehört das Erzeugen von Objektreferenzen und die Vermittlung von Methodenaufrufen an Objekte. Damit ist der Broker ideal geeignet für die Realisierung der unterschiedlichen Lastverteilungsmechanismen und die Gewährleistung der Migrations- und der Replikationstransparenz. Wie bereits in Abschnitt 4.2.1 erwähnt wurde, ermöglicht die Integration in den Broker auch Transparenz auf der Server-Seite, falls geeignete Persistenzmechanismen vorhanden sind.

Sowohl die Lasterfassung als auch die Lastverteilung werden in das verteilte Ablaufsystem integriert, da es die benötigte Information und Funktionalität zur Verfügung stellt und eine transparente Realisierung ermöglicht. Neben diesen konzeptionellen Überlegungen hängt die Implementierung des Lastverwaltungskonzepts auch von speziellen technischen Eigenschaften des zugrundeliegenden verteilten objektorientierten Systems ab, die an dieser Stelle nicht betrachtet werden.

4.2.6 Die Lastbewertung

Die Eigenschaften der Lastbewertung hängen teilweise von den Zielen der Lastverwaltung und der verfolgten Strategie ab. Einige Anforderungen an die Lastbewertung können jedoch aus den grundlegenden Charakteristika verteilter objektorientierter Systeme abgeleitet werden.

Verteilte objektorientierte Systeme können eng- oder lose gekoppelt sein. Eng gekoppelte Systeme sind in der Regel auf ein lokales Netz beschränkt, wohingegen lose gekoppelte Systeme durch ein Weitverkehrsnetz verbunden sind. Die örtliche Ausdehnung reicht von einem lokalen Netz mit einer begrenzten Anzahl an Rechnern bis hin zu weltweit verteilten Systemen mit sehr vielen Rechnern. Insbesondere bei lose gekoppelten Systemen muß besonderes Augenmerk auf die Skalierbarkeit der Lastbewertung gerichtet werden, da die Leistungsfähigkeit einer einzelnen Komponente beschränkt ist, d.h. sie kann nur eine begrenzte Anzahl von Rechnern und Objekten verwalten. Während bei eng gekoppelten Systemen eine zentrale Lastbewertungskomponente eingesetzt werden kann, muß bei lose gekoppelten Systemen auf gruppierte oder lokale Komponenten ausgewichen werden, um die Skalierbarkeit des Systems zu gewährleisten.

Die Wahl der Lokalisation legt auch den Wirkungsbereich und die Systemkenntnis fest. Bei eng gekoppelten Systemen mit einer zentralen Komponente ist der Wirkungs-

bereich stets maximal und die Systemkenntnis global. Dies ist auch in Bezug auf die Skalierbarkeit der Lastbewertung akzeptabel, da eng gekoppelte Systeme meist nur über eine begrenzte Anzahl von Rechnern und Objekten verfügen. Gruppierte und lokale Lastbewertungskomponenten, die insbesondere bei lose gekoppelten Systemen zum Einsatz kommen, haben in der Regel einen minimalen oder begrenzten Wirkungsbereich, sowie lokale oder partielle Systemkenntnis, um die Skalierbarkeit des Systems sicherzustellen.

Lokale und gruppierte Lastbewertungskomponenten sollten stets kooperieren, da sonst der Lastausgleich als Minimalziel der Lastverwaltung nicht erreicht werden kann. Beispielsweise kann es bei partieller Systemkenntnis und begrenztem Wirkungsbereich vorkommen, daß die Wirkungsbereiche zweier Lastbewertungskomponenten disjunkt sind. Befindet sich in einem dieser Wirkungsbereiche ein überlasteter Rechner und im anderen ein unterlasteter Rechner, dann kann die Lastbewertung keinen Lastausgleich herbeiführen. Auch überlappende Wirkungsbereiche lösen dieses Problem nur teilweise, da nicht sichergestellt werden kann, daß sich alle Wirkungsbereiche paarweise überlappen. Lediglich die Kooperation der Lastbewertungskomponenten gewährleistet, daß der Lastausgleich herbeigeführt werden kann.

Die Informationsgrundlage der Lastbewertung muß dynamisch sein, da bei statischer Lastbewertung keine Lasterfassung vorhanden ist, wodurch das Minimalziel der Lastverwaltung in der Regel nicht erreicht werden kann. Auch die Adaptivität der Lastbewertung ist eine wichtige Forderung. Sie beschreibt die Fähigkeit, sich an eine veränderte Lastsituation anzupassen, die nicht durch lastverwaltete Anwendungen hervorgerufen wurde. Dies ist jedoch die Grundvoraussetzung zur Behandlung von Hintergrundlast, die in verteilten objektorientierten Systemen eine wichtige Rolle spielt.

4.2.7 Das Lastverwaltungskonzept im Überblick

Das Konzept zur Lastverwaltung in verteilten objektorientierten Systemen, das in den vorangegangenen Abschnitten vorgestellt wurde, ist in Tabelle 4.1 im Überblick dargestellt.

Lastverteilungseinheiten sind Objekte, da diese sowohl feingranulare als auch transparente Lastverteilung ermöglichen. Als Ausführungseinheiten dienen folglich Server-Rechner und Server-Prozesse. Eine wesentliche Innovation dieses Konzepts ist die kombinierte Anwendung mehrerer Lastverteilungsmechanismen. Die Initialplazierung weist neu erzeugten Objekten geeignete Rechner zu. Die Migration paßt die Entscheidung der Initialplazierung zur Laufzeit der Objekte an eine veränderte Lastsituation an. Bei extremer Überlast kann mit Hilfe der Replikation die Granularität der Lastverteilung zusätzlich erhöht werden. Die Lastverteilung ist für den Entwickler transparent, da sie ist in das Ablaufsystem integriert ist.

Die Lasterfassung muß wegen der Schichtung verteilter objektorientierter Systeme sowohl die Ressourcen des Rechensystems als auch die Ressourcen der Anwendung betrachten. Auf Anwendungsebene wird Lastinformation über Objekte und auf der Ebene des Rechensystems Lastinformation über Rechner und deren Kommunikationsnetzwerk betrachtet. Um den Zugriff auf alle Ebenen eines verteilten Systems zu

Tabelle 4.1 Klassifikationsmerkmale eines Lastverwaltungskonzepts für verteilte objektorientierte Systeme

Komponente	Klassifikationskriterium	Varianten	
Lasterfassung	Ebenen	Rechensystem und Anwendung	
	Ressourcen	abhängig von der Lastverwaltungsstrategie	
	Integration	Ablaufsystem	
	Transparenz	transparent	
Lastbewertung	Ziel	abhängig von der Lastverwaltungsstrategie	
	Lokalisation	lokal oder gruppiert	zentral
	Interaktion der Last−bewertungskomponenten	kooperierend	−−
	Systemkenntnis	lokal oder partiell	global
	Wirkungsbereich	minimal oder begrenzt	maximal
	Informationsgrundlage	dynamisch	
	Adaptivität	adaptiv	
	Steuerung und Initiierung	abhängig von der Implementierung	
Lastverteilung	Lastverteilungseinheiten	Objekte	
	Unterbrechbarkeit der Lastverteilungseinheiten	bedingt−präemptiv	
	Ausführungseinheiten	Server−Prozesse und Server−Rechner	
	Mechanismen	Lastzuweisung und Lastverschiebung	
	Lastzuweisung	Initialplazierung	
	Lastverschiebung	Migration und Replikation	
	Integration	Ablaufsystem	
	Transparenz	transparent	

ermöglichen, ist die Lasterfassung in das Ablaufsystem integriert. Dies gewährleistet zudem die Transparenz der Lasterfassung, da kein Mehraufwand für den Entwickler entsteht.

Bei der Lastbewertung muß zwischen eng- und lose gekoppelten Systemen unterschieden werden. Für eng gekoppelte Systeme empfiehlt das Lastverwaltungskonzept eine zentrale Lastbewertungskomponente mit globaler Systemkenntnis und maximalem Wirkungsbereich. In lose gekoppelten Systemen sollten lokale oder gruppierte Lastbewertungskomponenten mit lokaler oder partieller Systemkenntnis, sowie minimalem oder begrenztem Wirkungsbereich verwendet werden, um die Skalierbarkeit zu gewährleisten. Darüber hinaus muß die Lastbewertung stets dynamisch sein, um sich einer veränderten Lastsituation anpassen zu können. Schließlich wird noch die Ad-

aptivität der Lastbewertung gefordert, damit die Lastverwaltung auf Hintergrundlast reagieren kann.

4.3 Migration und Replikation

Die vorangegangenen Abschnitte beschreiben ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen. Kernpunkt dieses Konzepts ist die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen, insbesondere der Migration und der Replikation. Die Migration und die Replikation sind jedoch nicht in jedem Fall anwendbar. Deshalb werden im folgenden Objekte und ihre Eigenschaften näher untersucht, um detaillierte Aussagen über die Migrierbarkeit und die Replizierbarkeit treffen zu können.

4.3.1 Identität und Äquivalenz von Objekten

Ein zentrales Konzept verteilter objektorientierter Systeme ist die Verwendung von Objektreferenzen. Eine Objektreferenz ist ein eindeutiger Identifikator für ein Objekt. Die Adressierung von Objekten geschieht über den Broker, der Objektreferenzen auf Objekte abbildet. Diese Abbildung ist surjektiv[1] aber nicht injektiv[2], da unterschiedliche Objektreferenzen auf dasselbe physische Objekt verweisen können [98]. Damit ist die Abbildung nicht bijektiv[3] und auch nicht umkehrbar. Folglich wird die Identität von Objekten ausschließlich durch die Objekte selbst und nicht durch ihre Objektreferenzen bestimmt [124].

Definition 4.3 *Objektidentität*

> Objekte sind identisch wenn es sich um dasselbe physische Objekt handelt. □

Die Identität ist ein sehr starker Begriff, mit dem die Replikation nicht beschrieben werden kann. Bei der Replikation entstehen sogenannte Replikate, die nicht identisch sind, da es sich um physisch unterschiedliche Objekte handelt. Die Suche nach einer schwächeren Beschreibung der Ähnlichkeit von Objekten führt zum Begriff der Objektäquivalenz:

Definition 4.4 *Objektäquivalenz*

> Objekte sind äquivalent wenn sie identische Schnittstellen haben und wenn zu jedem Zeitpunkt t gilt: Die Objekte würden zum Zeitpunkt t für dieselbe Anfrage ein semantisch identisches Ergebnis liefern. □

Diese Definition der Objektäquivalenz beruht auf der Verhaltensgleichheit äquivalenter Objekte und wird deshalb auch als Verhaltensäquivalenz bezeichnet. Das Verhalten

[1] Eine Funktion $f : \mathcal{A} \rightarrow \mathcal{B}$ ist surjektiv, wenn sie jedes Element der Menge $\mathcal{B}$ trifft.
[2] Eine Funktion $f : \mathcal{A} \rightarrow \mathcal{B}$ ist injektiv, wenn sie jedes Element von $\mathcal{B}$ höchstens einmal trifft.
[3] Eine Funktion ist bijektiv oder umkehrbar, wenn sie surjektiv und injektiv ist.

eines Objektes ist durch die Ergebnisse charakterisiert, mit welchen es Anfragen beantwortet. Objekte sind äquivalent, wenn sie zu jedem Zeitpunkt für dieselbe Anfrage ein semantisch identisches Ergebnis liefern würden. Damit sind äquivalente Objekte aus der Sicht eines Clients jederzeit austauschbar. Der Begriff semantische Identität besagt, daß sich der Wert der Ergebnisse unterscheiden kann, die Bedeutung für die Clients jedoch identisch ist. Bei deterministischen Objekten kann diese Unterscheidung in der Regel entfallen, so daß semantische Identität als Wertegleichheit interpretiert werden kann. Bei nicht deterministischen Objekten hingegen steht die Bedeutungsgleichheit im Vordergrund. Dies sei am Beispiel eines Objekts erläutert, das einen Zufallszahlengenerator realisiert. Dieses Objekt liefert für identische Anfragen unterschiedliche Werte, nämlich Zufallszahlen und ist somit nicht deterministisch. Aus der Sicht eines Clients ist die Zufallsverteilung der angeforderten Zahlen von Bedeutung; die Werte selbst spielen eine untergeordnete Rolle. Damit sind zwei Zufallszahlengeneratoren durchaus äquivalent, da die gelieferten Werte semantisch identisch sind. Daraus ergibt sich ein wesentlicher Vorteil dieser Definition: Sie ist sowohl für deterministische als auch für nicht deterministische Objekte geeignet.

Basierend auf der Objektäquivalenz können nun auch Replikate beschrieben werden. Äquivalente Objekte sind aus der Sicht eines Clients jederzeit austauschbar, da sie zu jedem Zeitpunkt für dieselbe Anfrage ein semantisch identisches Ergebnis liefern würden. Diese Forderung muß wegen der Replikationstransparenz auch für Replikate gelten.

Definition 4.5 *Replikate*

Replikate sind physisch verschiedene aber äquivalente Objekte. □

Die Objektäquivalenz ist eine Äquivalenzrelation, da sie, wie leicht zu sehen ist, reflexiv[4], symmetrisch[5] und transitiv[6] ist. Als Äquivalenzrelation zerlegt sie die Menge aller Objekte in Äquivalenzklassen, sogenannte Replikatgruppen.

Definition 4.6 *Replikatgruppe*

Die Replikatgruppe eines Objekts ist die Menge aller Objekte, die zu diesem Objekt äquivalent sind. □

Die Menge aller Replikate eines Objekts bildet seine Replikatgruppe. Da Replikate jederzeit austauschbar sind, können Clients anstelle eines Objekts jedes beliebige Element seiner Replikatgruppe anfragen.

Auch in anderen Bereichen der Informatik wird zwischen Objektidentität und Objektäquivalenz unterschieden. In objektorientierten Programmiersprachen sind Objekte identisch, wenn ihre Referenzen auf dieselbe Speicherzelle verweisen. Sie sind äquivalent, wenn der Inhalt der Speicherzellen, auf die ihre Referenzen zeigen, identisch ist [50]. Im Bereich der Datenbanktechnik haben sich KHOSHAFIAN und COPELAND

[4]Eine Relation $\circ$ über der Menge $\mathcal{A}$ ist reflexiv, wenn gilt: $a \circ a, \forall a \in \mathcal{A}$.

[5]Eine Relation $\circ$ über der Menge $\mathcal{A}$ ist symmetrisch, wenn gilt: $a \circ b \Rightarrow b \circ a, \forall a, b \in \mathcal{A}$.

[6]Eine Relation $\circ$ über der Menge $\mathcal{A}$ ist transitiv, wenn gilt: $a \circ b \vee b \circ c \Rightarrow a \circ c, \forall a, b, c \in \mathcal{A}$.

[60] mit der Identität und der Äquivalenz von Objekten befaßt. Analog zu der hier verwendeten Definition bezeichnen sie Objekte als identisch, wenn es sich um dasselbe Objekt handelt. Bei der Äquivalenz unterscheiden sie zwischen flacher und tiefer Äquivalenz. Objekte sind flach äquivalent, wenn sie vom gleichen Typ sind und ihre Attribute identisch sind. Sie werden als tief äquivalent bezeichnet, wenn zusätzlich auch alle referenzierten Objekte äquivalent sind. Diese Beispiele zeigen, daß in unterschiedlichen Bereichen der Informatik ähnliche Definitionen für die Objektidentität verwendet werden. Für die Beschreibung der Objektäquivalenz gibt es zwei konkurrierende Ansätze: Die Verhaltens- und die Zustandsäquivalenz. Die Verhaltensäquivalenz entspricht der hier verwendeten Definition. Im Gegensatz dazu beschreibt die Zustandsäquivalenz die Ähnlichkeit von Objekten anhand ihres Zustandes. Im folgenden Abschnitt werden beide Vorgehensweisen verglichen und ihre Tauglichkeit für verteilte objektorientierte Systeme diskutiert.

4.3.2 Der Zustand eines Objekts

Objekte sind gekennzeichnet durch ihr Verhalten und ihren Zustand. Das Verhalten wird durch die Schnittstelle und ihre Methoden ausgedrückt, mit deren Hilfe Objekte dazu veranlaßt werden, bestimmte Aktionen auszuführen. Der Zustand umfaßt die Attribute eines Objekts und gegebenenfalls auch die Attribute aller referenzierten Objekte.

Definition 4.7 *Zustand*

> Als Zustand eines Objekts bezeichnet man die Menge aller Attribute des Objekts.
>
> Der rekursive Zustand eines Objekts ist die Vereinigungsmenge seines Zustandes und des rekursiven Zustandes aller referenzierten Objekte. □

Zwischen dem Zustand und der Implementierung besteht ein enger Zusammenhang [58]. Zur Erläuterung wird das Beispiel eines Kreises herangezogen. Dieser Kreis sei wahlweise durch zwei Objekte repräsentiert, die über identische Schnittstellen verfügen, mit deren Hilfe die Kreisfläche abgefragt werden kann. Beide Objekte implementieren die Berechnung der Kreisfläche auf unterschiedliche Art und Weise: Ein Objekt berechnet die Fläche aus dem Radius, das andere aus dem Durchmesser. Folglich verfügt ein Objekt über das Attribut `Radius` und das andere über das Attribut `Durchmesser`. Die Implementierung eines Objekts entscheidet also über seine Attribute und damit auch über seinen Zustand.

Die Abhängigkeit des Zustandes von der Implementierung hat Auswirkungen auf die Objektäquivalenz. Vergleicht man die beiden Kreis-Objekte anhand ihres Zustandes, so wären sie nicht äquivalent, da sich ihre Attribute und deren Werte unterscheiden. Betrachtet man jedoch das Verhalten der Objekte, dann sind sie äquivalent, da sie bei der Abfrage der Kreisfläche stets dasselbe Ergebnis liefern. Zustands- und Verhaltensäquivalenz sind nicht nur verschiedene Betrachtungsweisen, sie liefern auch unterschiedliche Ergebnisse. Bei verteilten objektorientierten Systemen ist die Verhaltensäquivalenz die geeignete Sichtweise, da sie dem intuitiven Verständnis der Äquivalenz

entspricht. Dies ist am Beispiel der Kreis-Objekte zu sehen, die beide denselben Kreis repräsentieren und somit auch als äquivalent betrachtet werden sollten.

Obwohl sich der Zustand nicht zur Beschreibung der Äquivalenz eignet, ist er für die Lastverschiebung und insbesondere die Replikation von großer Bedeutung. Objekte zerfallen nämlich in zwei Klassen: Zustandslose und zustandsbehaftete Objekte.

Definition 4.8 *Zustandslose und zustandsbehaftete Objekte*

> Ein Objekt ist zustandslos, wenn es über keine veränderlichen Attribute verfügt. Anderenfalls ist es zustandsbehaftet. □

Ein Beispiel für ein zustandsbehaftetes Objekt ist ein Namensdienst, der die Abbildung intuitiver Namen auf Adressen oder Referenzen ermöglicht. Ein entsprechender Dienst muß bei der Registrierung den Namen und die dazugehörende Referenz in geeigneten Attributen speichern und ist somit zustandsbehaftet. Das Network File System (NFS), das schon in Kapitel 2.3.1 erwähnte wurde, ist ein zustandsloser Dienst [137]. NFS ist ein verteiltes Dateisystem, das sämtliche Dateizugriffe über sogenannte Dateideskriptoren abwickelt. Ein Deskriptor beinhaltet sowohl den Dateinamen als auch die Position innerhalb der Datei, ab der gelesen oder geschrieben werden soll. Aus diesem Grund benötigt der NFS-Server keine Attribute, um Information über offene Dateien zwischen einzelnen Aufrufen zu speichern und ist damit zustandslos.

Bei der Replikation von zustandsbehafteten Objekten sind spezielle Protokolle nötig, um den Zustand der Replikate konsistent zu halten [74] und damit ihre Äquivalenz zu gewährleisten. Würde man beispielsweise einen Namensdienst replizieren, dann müßte jeder Name, der bei einem Replikat registriert wird auch allen anderen mitgeteilt werden, damit er bei allen Replikaten abrufbar ist. Der Mehraufwand, den dieses Konsistenzprotokoll verursacht, macht die erzielte Leistungssteigerung wieder zunichte. Effiziente Konsistenzprotokolle konnten bisher nur auf Anwendungsebene entwickelt werden [76, 127]. Die Realisierung auf der Ebene der Anwendung widerspricht jedoch der Forderung nach transparenter Lastverteilung. Zustandslose Objekte hingegen haben den Vorteil, daß die Replikate keine Konsistenzprotokolle benötigen. Dies ist auch der Grund dafür, daß die Replikation bisher nur bei zustandslosen Objekten angewandt wird.

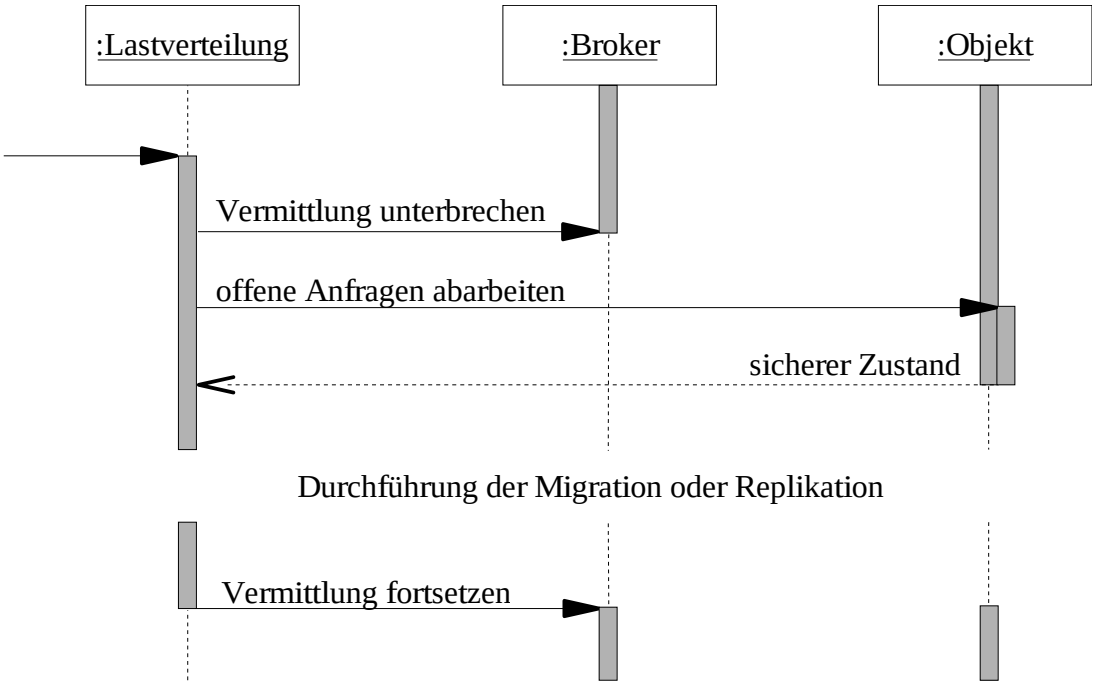
4.3.3 Der sichere Zustand

Bei der Migration und der Replikation muß gewährleistet sein, daß keine inkonsistenten Zustände bei den beteiligten Objekten entstehen. Zu einem inkonsistenten Zustand kommt es, wenn bei der Lastverschiebung Anfragen oder Ergebnisse vervielfältigt werden oder verlorengehen. Um dies zu verhindern, werden die an der Lastverschiebung beteiligten Objekte in einen sicheren Zustand versetzt.

Definition 4.9 *Sicherer Zustand*

> Ein Objekt befindet sich in einem sicheren Zustand, wenn keine Anfragen abgearbeitet werden und keine Anfragen zur Bearbeitung anstehen. □

Abbildung 4.1 Protokoll zum Erzeugen eines sicheren Zustandes



Wie bereits in Abschnitt 4.2.1 beschrieben wurde, setzen auch die meisten Persistenzmechanismen einen sicheren Zustand voraus, da die Information über aktive und anstehende Anfragen in den Servern und nicht in den Objekten gespeichert ist und damit von den Persistenzmechanismen nicht erfaßt wird.

Die Lastverteilungskomponente muß ein geeignetes Protokoll zur Verfügung stellen, das es ermöglicht, einen sicheren Zustand herzustellen. Dieses Protokoll ist in Abbildung 4.1 als UML-Sequenzdiagramm[7] dargestellt. Zuerst veranlaßt die Lastverteilungskomponente den Broker, die Vermittlung von Anfragen für das betreffende Objekt zu unterbrechen. Dadurch werden neue Anfragen solange verzögert, bis die Lastverschiebeaktion abgeschlossen ist. Anschließend wartet die Lastverteilung bis das Objekt alle anstehenden Anfragen abgearbeitet und damit einen sicheren Zustand erreicht hat. Um schneller in einen sicheren Zustand zu gelangen, kann das Objekt alle Anfragen, mit deren Abarbeitung noch nicht begonnen wurde, an den Broker zurückgeben. Anschließend wird die eigentliche Lastverschiebeaktion, d.h. die Migration oder die Replikation, ausgeführt. Zum Schluß benachrichtigt die Lastverteilung den Broker, daß er die Vermittlung der Anfragen wieder aufnehmen kann.

Dieses Protokoll ermöglicht es, Objekte zwischen aufeinanderfolgenden Anfragen zum Zweck der Lastverteilung zu unterbrechen. Eine Unterbrechung ist also nur ein-

[7] UML (Unified Modeling Language) ist eine grafische Beschreibungssprache zur Modellierung von Software-Systemen [87].

geschränkt möglich, weshalb Objekte in Abschnitt 4.2.1 als bedingt-präemptiv bezeichnet wurden. Diese Aussage unterliegt jedoch einer Einschränkung:

Definition 4.10 *Unterbrechbarkeit von Objekten*

> Ein Objekt ist bedingt-präemptiv, wenn keine seiner Methoden direkt oder indirekt Methoden desselben Objekts aufruft. Anderenfalls ist es nicht-präemptiv. □

Ein sicherer Zustand kann, nach dem oben beschriebenen Protokoll, nur bei bedingt-präemptiven Objekten hergestellt werden. Nicht-präemptive Objekte stellen Anfragen an sich selbst. Diese Anfragen würden bei einer Unterbrechung der Vermittlungstätigkeit des Brokers nicht an das Objekt weitergeleitet, was schließlich eine Verklemmung zur Folge hätte. Direkte Selbstanfragen können vermieden werden, indem sie nicht über den Broker sondern als lokale Anfragen ausgeführt werden. Eine indirekte Selbstanfrage führt über mindestens ein weiteres Objekt und kann somit auch nicht durch eine lokale Anfrage ersetzt werden. Generell sind Selbstanfragen jedoch sehr selten, da sie nur bei Objekten mit nebenläufiger Abarbeitung verwendet werden können. Bei sequentieller Abarbeitung führt jede Selbstanfrage zur Verklemmung, da sie erst dann bearbeitet werden würde, wenn die ursprüngliche Anfrage beendet ist.

4.3.4 Migrierbarkeit und Replizierbarkeit

Die Migration und die Replikation sind an bestimmte Vorbedingungen gebunden. Anhand des bisher gebildeten Begriffsgerüst können nun die Eigenschaften von Objekten identifiziert werden, die über die Anwendbarkeit der Lastverschiebung entscheiden.

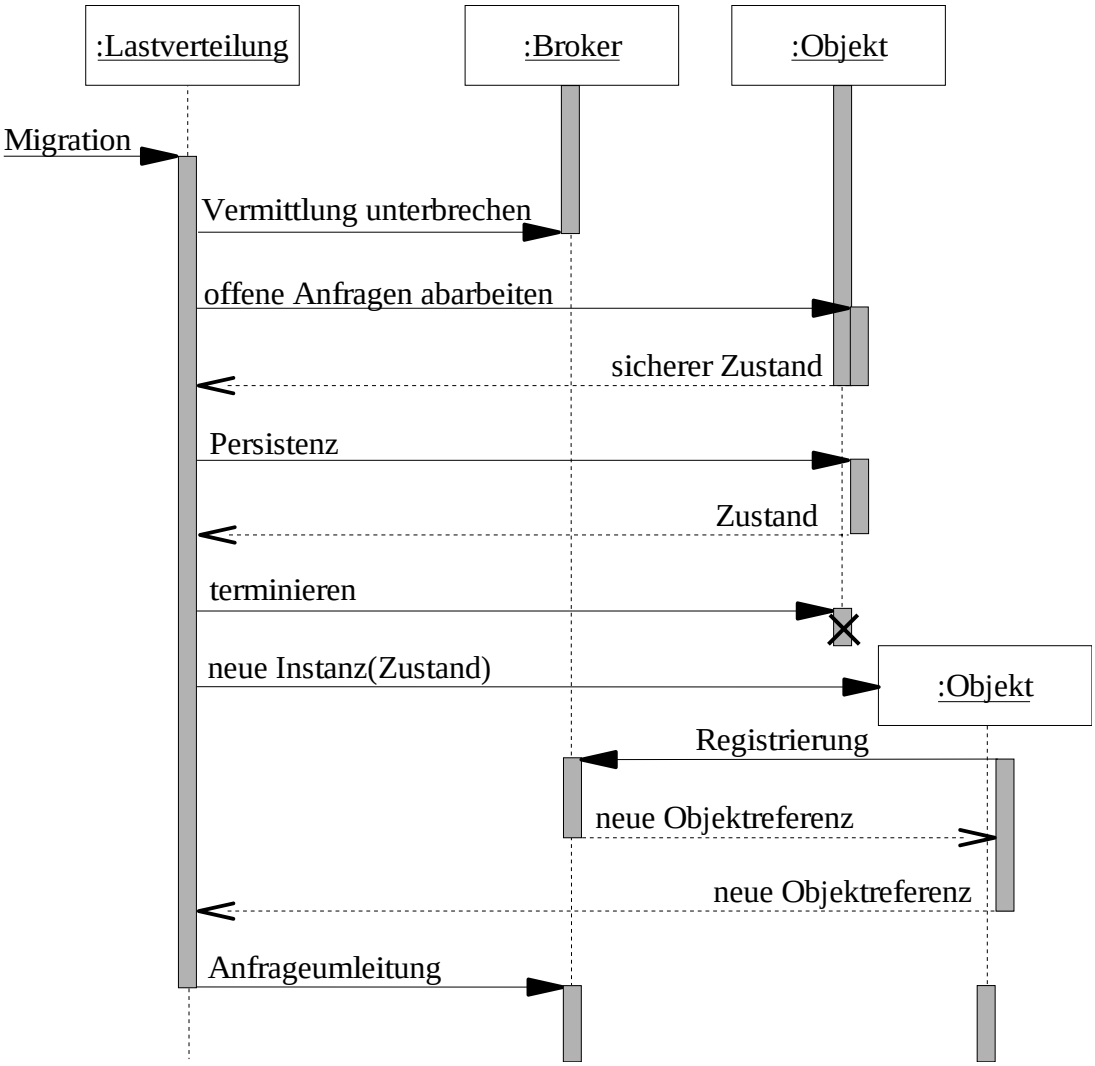
Definition 4.11 *Migrierbarkeit*

> Ein Objekt ist migrierbar, wenn es ein Verfahren gibt, welches das Objekt durch ein äquivalentes Objekt ersetzt. □

Wie anhand dieser Definition zu sehen ist, hängt die Migrierbarkeit von Objekten von den verfügbaren Migrationsverfahren ab. In der Praxis sind unterschiedliche Verfahren vorstellbar. Im folgenden wird ein Migrationsverfahren vorgestellt, das alle konzeptionellen Anforderungen aus Abschnitt 4.2.7 berücksichtigt. Das Verfahren ist so allgemein wie möglich gehalten, um ein breites Spektrum realer Systeme abdecken zu können. Anschließend können anhand dieses Migrationsverfahrens exemplarisch die Vorbedingungen für die Migrierbarkeit von Objekten abgeleitet werden.

Das hier gewählte Verfahren basiert auf dem im vorangegangenen Abschnitt vorgestellten Protokoll zur Herstellung eines konsistenten Zustandes. Abbildung 4.2 zeigt das Ablaufprotokoll für die Migration als UML-Sequenzdiagramm. Zuerst versetzt die Lastverteilung das zu migrierende Objekt in einen sicheren Zustand. Anschließend wird mit Hilfe eines geeigneten Persistenzmechanismus der Zustand des Objekts extrahiert. Der Zustand enthält alle Attribute des Objekts, was es dem Persistenzmechanismus erlaubt, ein neues Objekt daraus zu erzeugen. Damit wird das ursprüngliche Objekt überflüssig und kann entfernt werden. Die Lastverteilung erzeugt dann mit

Abbildung 4.2 Protokoll zur Migration eines Objekts



Hilfe des vorgegebenen Persistenzmechanismus eine neue Instanz des Objekts und parametrisiert diese mit dem Zustand des ursprünglichen Objekts. Daraufhin registriert sich das neue Objekt beim Broker und erhält eine Objektreferenz, die anschließend der Lastverteilung mitgeteilt wird. Zum Schluß informiert die Lastverteilung den Broker, daß Anfragen mit der ursprünglichen Objektreferenz auf das neue Objekt umgeleitet werden müssen.

Bei Verwendung dieses Protokolls entscheiden insbesondere drei Faktoren darüber, ob die Migration durchführbar ist, d.h. ob es möglich ist ein äquivalentes Objekt zu erzeugen:

- Das betreffende Objekt muß bedingt-präemptiv sein, damit ein sicherer Zustand hergestellt werden kann.

- □ Es muß ein geeigneter Persistenzmechanismus vorhanden sein, mit dem der Zustand des Objekts extrahiert und daraus ein neues Objekt erzeugt werden kann.

- □ Das neu erzeugte Objekt muß äquivalent zum ursprünglichen Objekt sein, unter der Voraussetzung, daß das ursprüngliche Objekt zuvor entfernt wird.

Die Wahl eines Persistenzmechanismus hängt, wie in Abschnitt 4.2.1 beschrieben, vom verwendeten verteilten objektorientierten System ab. Bei zustandslosen Objekten ist kein Persistenzmechanismus nötig, da in diesem Fall auch keine Zustandsübertragung stattfindet. Die Forderung, daß das ursprüngliche und das neu erzeugte Objekt äquivalent sein müssen, ist bei der Migration in der Regel trivial. Wenn die Implementierung beider Objekte identisch ist, dann ist die Äquivalenz stets gegeben, da auch ihr Zustand übereinstimmt [117]. Wenn unterschiedliche Implementierungen eines Objekts existieren, muß der Entwickler gewährleisten, daß sie sich äquivalent verhalten. Mehrere Implementierungen eines Objekts sind dann sinnvoll, wenn unterschiedliche Betriebssysteme oder Programmiersprachen unterstützt werden sollen. Beispielsweise könnte eine Implementierung in Java realisiert sein, um die Lauffähigkeit auf unterschiedlichen Rechnern und Betriebssystemen zu gewährleisten. Eine andere Implementierung könnte ein für ein spezielles Zielsystem übersetztes C++ Programm sein, das eine höhere Performanz erreicht. In diesem Fall muß ein Persistenzmechanismus gewählt werden, der sowohl verschiedene Rechner und Betriebssysteme als auch unterschiedliche Programmiersprachen unterstützt.

Analog zur Migrierbarkeit ist auch die Replizierbarkeit durch die Konstruktion eines äquivalenten Objekts definiert.

Definition 4.12 *Replizierbarkeit*

> Ein Objekt ist replizierbar, wenn es ein Verfahren gibt, mit dem ein Replikat erzeugt werden kann. □

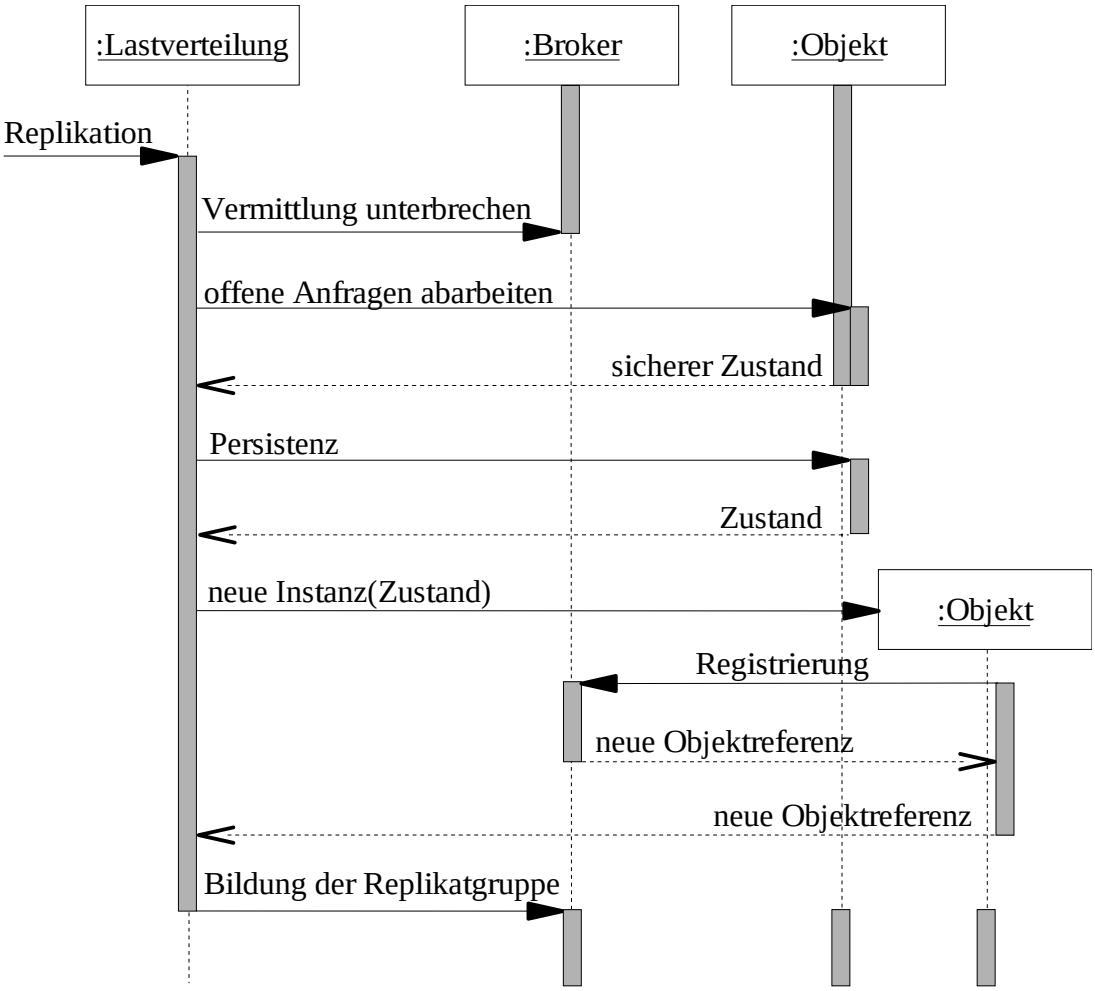
Das hier vorgestellte Verfahren zur Replikation ist dem der Migration sehr ähnlich. Das entsprechende Ablaufprotokoll ist in Abbildung 4.3 dargestellt. Im Gegensatz zur Migration wird das ursprüngliche Objekt nicht entfernt, so daß zwei Replikate entstehen. Die Lastverteilung faßt die Replikate zu einer Replikatgruppe zusammen und registriert diese beim Broker. Dadurch wird der Broker veranlaßt, Anfragen an die Replikatgruppe unter den einzelnen Replikaten aufzuteilen.

Auch die Anwendung der Replikation ist an einige Vorbedingungen gebunden. Im einzelnen müssen folgende Forderungen erfüllt sein, damit ein Objekt der Replikation unterzogen werden kann:

- □ Das betreffende Objekt muß bedingt-präemptiv sein, damit ein sicherer Zustand hergestellt werden kann.

- □ Es muß ein geeigneter Persistenzmechanismus vorhanden sein, mit dem der Zustand des Objekts extrahiert und daraus ein neues Objekt erzeugt werden kann.

Abbildung 4.3 Protokoll zur Replikation eines Objekts



- Das neu erzeugte Objekt muß äquivalent zum ursprünglichen Objekt sein, unter der Voraussetzung, daß das ursprüngliche Objekt weiterhin existiert. Dies ist der Fall, wenn eine der folgenden Bedingung erfüllt ist:
  - Das Objekt ist zustandslos.
  - Das Objekt verfügt über einen redundanten Zustand.

Im Gegensatz zur Migration wird bei der Replikation das ursprüngliche Objekt nicht entfernt. Es muß also gewährleistet sein, daß die entstehenden Replikate äquivalent, d.h. für Clients jederzeit austauschbar sind. Bei zustandslosen Objekten ist dies stets der Fall, da sie keine Attribute haben und somit über keinerlei Speicherfähigkeit verfügen. Die Speicherfähigkeit ist jedoch die Grundvoraussetzung dafür, daß sich das Verhalten eines Objekts ändern kann. Zustandsbehaftete Objekte können nur repliziert werden, wenn ihr Zustand redundant ist.

Definition 4.13 *Redundanter Zustand*

Der Zustand eines Objekts ist redundant, wenn seine Äquivalenzbeziehung zu anderen Objekten durch Zustandsänderungen nicht beeinflußt wird. □

Ein redundanter Zustand gewährleistet, daß die Äquivalenz eines Objekts und seiner Replikate bei einer Veränderung des Zustandes stets erhalten bleibt. Damit sind Objekte mit redundantem Zustand per Definition replizierbar, ohne daß ein explizites Konsistenzprotokoll nötig wäre. In der Praxis gibt es mehrere Ursachen für das Auftreten von redundanten Zuständen:

- Im einfachsten Fall werden die Attribute eines Objekts nie verändert, d.h. nach der Initialisierung finden nur lesende Zugriffe statt.
- Die Attribute dienen als Zwischenspeicher für konstante Werte (Read-Only-Cache), die entweder anderweitig beschafft oder berechnet werden können.
- Ein implizites Konsistenzprotokoll gewährleistet die Äquivalenz. Implizite Konsistenzprotokolle werden häufig bei Read-Write-Caches eingesetzt, um die Konsistenz verteilter Zwischenspeicher sicherzustellen.

In verteilten objektorientierten Systemen tauchen redundante Zustände sehr häufig auf. Insbesondere Zwischenspeicher werden in verteilten Systemen oft eingesetzt, um kostspielige entfernte Zugriffe durch kostengünstige lokale Speicherzugriffe zu ersetzen.

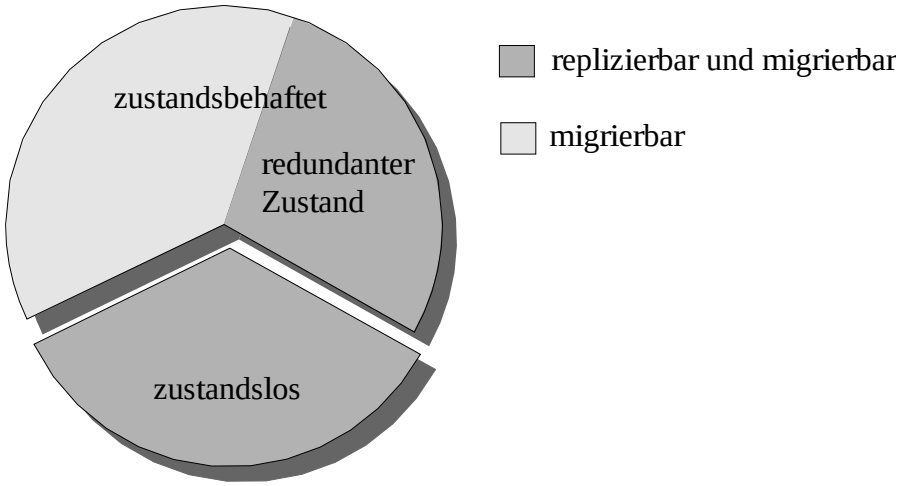
Ein Beispiel für einen Read-Write-Cache mit implizitem Konsistenzprotokoll ist ein WWW-Proxy. Dieser dient als Zwischenspeicher für WWW-Clients, die den WWW-Proxy anstelle eines WWW-Servers anfragen. Der Proxy prüft, ob die gewünschte Seite bereits in seinem Zwischenspeicher vorhanden ist und fragt sie gegebenenfalls beim WWW-Server an. Folglich liefert ein WWW-Proxy für jede Anfrage die entsprechende Seite, unabhängig davon, ob diese gespeichert ist oder nicht. Deshalb hat der Zustand keine Auswirkung auf das Verhalten und ist somit redundant. Wie bereits erwähnt handelt es sich bei einem WWW-Proxy um einen Read-Write-Cache. Aus diesem Grund verfügen Proxies über ein implizites Konsistenzprotokoll mit dem Änderungen gespeicherter Seiten aktualisiert werden. Dabei handelt es sich um ein schwaches Konsistenzprotokoll, was dazu führen kann, daß Proxies kurzzeitig veraltete Seiten zwischenspeichern. Insofern können verschiedene Proxies dieselbe Anfrage mit inhaltlich unterschiedlichen Seiten beantworten. Die gelieferten Ergebnisse sind dennoch semantisch identisch, da die inhaltlichen Unterschiede durch das WWW-Protokoll selbst bedingt sind. Dadurch ist die Äquivalenz unterschiedlicher Proxies gewährleistet.

Diese Betrachtungsweise der Replizierbarkeit geht weit über die bisher übliche hinaus. Bisher wurden nur zustandslose Objekte der Replikation unterzogen. Die Ausdehnung auf Objekte mit redundantem Zustand vergrößert die Menge der replizierbaren Objekte maßgeblich. Mit Hilfe des redundanten Zustandes wird eine neue Klasse von

Objekten definiert, die ohne Anwendung eines expliziten Konsistenzprotokolls replizierbar sind. Diese Einschränkung ist nötig, da explizite Konsistenzprotokolle bisher nur auf Anwendungsebene realisiert werden können, was transparente Lastverteilung unmöglich macht. Im Gegensatz dazu zählen Objekte mit implizitem Konsistenzprotokoll zur Klasse der Objekte mit redundantem Zustand. Ein implizites Konsistenzprotokoll ist fester Bestandteil einer Anwendung und wird nicht ausschließlich zum Zweck der Lastverschiebung aufgesetzt, wie es bei einem expliziten Konsistenzprotokoll der Fall ist. Dadurch bleibt die Transparenz der Lastverteilung von impliziten Konsistenzprotokollen unberührt.

Zusammenfassend kann man sagen, daß bedingt-präemptive Objekte stets migrierbar sind, vorausgesetzt es existiert ein geeigneter Persistenzmechanismus. Objekte sind ohne Anwendung eines expliziten Konsistenzprotokolls replizierbar, wenn sie darüber hinaus zustandslos sind oder über einen redundanten Zustand verfügen. Der Zusammenhang zwischen der Beschaffenheit des Zustandes und der Migrierbarkeit bzw. der Replizierbarkeit von Objekten ist in Abbildung 4.4 dargestellt.

Abbildung 4.4 Migrierbarkeit und Replizierbarkeit von Objekten



Die Eigenschaft der Migrierbarkeit und Replizierbarkeit muß vom Entwickler oder vom Administrator angegeben werden, da sie nicht zur Laufzeit bestimmbar ist. Eine wichtige Entscheidungshilfe dafür leisten die in diesem Abschnitt beschriebenen Kriterien zur Migrierbarkeit und Replizierbarkeit von Objekten. Die Einführung des redundanten Zustandes ermöglicht es, die Klasse der replizierbaren Objekte zu erweitern. Darüber hinaus bietet die gesonderte Betrachtung von Objekten mit redundantem Zustand weitere Vorteile hinsichtlich der Lastverteilung. So kann bei Objekten mit redundantem Zustand die Wirksamkeit von Lastverteilungsaktionen durch spezielle Optimierungen stark verbessert werden. Dieser Zusammenhang wird in den nächsten beiden Abschnitten näher erläutert.

4.3.5 Redundanter Zustand und Zustandsübertragung

Die Ablaufprotokolle zur Migration und zur Replikation, die bereits in den vorherigen Abschnitten erläutert wurden, enthalten beide einen Persistenzmechanismus, mit dem der Zustand des ursprünglichen Objekts auf das neu erzeugte Objekt übertragen werden kann. Die Zustandsübertragung ist jedoch nicht in jedem Fall zwingend notwendig. Zustandslose Objekte verfügen über keinen Zustand, so daß die Betrachtung der Objektpersistenz hinfällig ist. In diesem Fall benötigt man lediglich einen Mechanismus, mit dem eine neue Instanz des betrachteten Objekts erzeugt werden kann. Bei Objekten mit redundantem Zustand ist die Äquivalenz des Objekts und seiner Replikate stets gewährleistet; unabhängig von einer Veränderung des Zustandes. Da jeder Zustand letztendlich nur eine Variation des Initialzustandes ist, genügt es, neu erzeugte Objekte mit dem Initialzustand des Ursprungsobjekts zu versehen. Auch in diesem Fall kann auf einen Persistenzmechanismus verzichtet werden. Bei Objekten mit redundantem Zustand können jedoch die Kosten der Lastverschiebung durch die Zustandsübertragung erheblich reduziert werden. Beispielsweise gewährleistet die Zustandsübertragung, daß bei der Migration oder Replikation eines Zwischenspeichers alle gespeicherten Werte erhalten bleiben. Anderenfalls müßten diese berechnet oder neu beschafft werden, was erhebliche Zusatzkosten verursachen würde. In diesen Fällen kann die Zustandsübertragung die Kosten der Lastverschiebung erheblich verringern.

Die Zustandsübertragung ist bei Objekten mit redundantem Zustand dann sinnvoll, wenn die Kosten für die Rekonstruktion des Zustandes größer sind als die Übertragungskosten. Die Entscheidung, ob die Zustandsübertragung durchgeführt wird oder nicht, muß der Entwickler treffen. Schließlich kann nur der Entwickler die Kosten der Rekonstruktion und der Übertragung des Zustandes gegeneinander abwägen.

4.3.6 Statische und dynamische Bindung

Wie bereits in den vorangegangenen Abschnitten erwähnt wurde, teilt der Broker die Anfragen an eine Replikatgruppe zwischen den einzelnen Replikaten auf, um die Anfragelast auf alle Replikate zu verteilen. Aus der Sicht eines Clients stellt sich die Frage, wie der Broker aufeinanderfolgende Anfragen an eine bestimmte Replikatgruppe behandelt. Grundsätzlich gibt es zwei Möglichkeiten, wie der Broker Clients an Replikate binden kann:

Definition 4.14 *Bindung der Clients an Replikate*

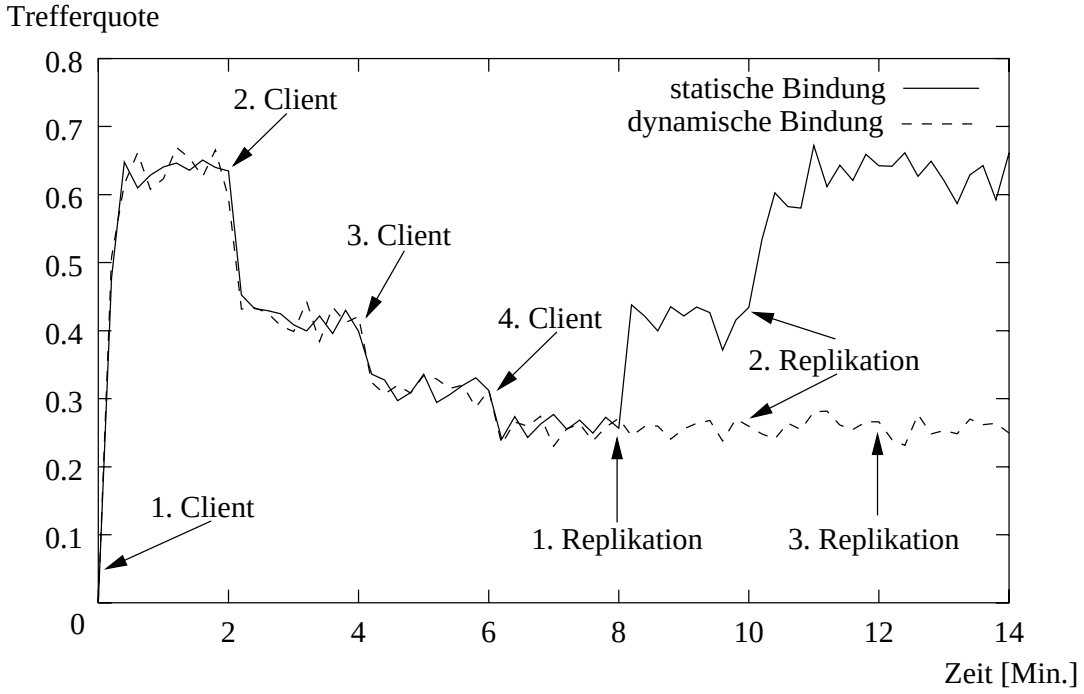
> Bei dynamischer Bindung kann der Broker aufeinanderfolgende Anfragen eines Clients unterschiedlichen Replikaten zuordnen.
>
> Man spricht von statischer Bindung, wenn einem Client ein Replikat fest zugeteilt ist, so daß aufeinanderfolgende Anfragen des Clients stets an dasselbe Replikat vermittelt werden. Bei der Lastverschiebung muß darüber hinaus gewährleistet sein, daß die Clients des Quellreplikats ausschließlich an die entsprechenden Zielreplikate gebunden werden. □

Der Broker muß also bei einer Migration mit statischer Bindung dafür Sorge tragen, daß alle Clients des Quellobjekts an das neu entstandene Zielobjekt gebunden werden. Bei der Replikation darf er die Clients des Quellreplikats ausschließlich zwischen dem Quellreplikat und dem neu entstandenen Zielreplikat aufteilen.

Die Entscheidung für eine der beiden Bindungsarten wirkt sich sowohl auf die Möglichkeiten der Lastverteilung als auch auf die erzielbare Leistungssteigerung aus. Die statische Bindung schränkt den Entscheidungsspielraum der Lastverteilung ein, da es nicht möglich ist, die Bindung während der Lebensdauer eines Clients zu verändern. Damit entfällt die Möglichkeit Clients innerhalb einer Replikatgruppe umzuverteilen. Dieses Problem tritt bei der dynamischen Bindung nicht auf. Die dynamische Bindung kann jedoch bei Objekten mit redundantem Zustand die Leistungssteigerung, die durch die Lastverteilung erzielt werden kann, erheblich verringern. Dieser Effekt tritt insbesondere bei Objekten auf, die einen Zwischenspeicher realisieren. Das entscheidende Leistungskriterium eines Zwischenspeichers ist die Trefferquote. Sie ist durch das Verhältnis der Anfragen, die aus dem Zwischenspeicher bedient werden und der Gesamtzahl der Anfragen definiert. Durch die Replikation erhöht sich die Größe des Zwischenspeichers der gesamten Replikatgruppe. Bei statischer Bindung führt dies in der Regel zu einer Erhöhung der Trefferquote, da die Anzahl der Clients pro Replikat sinkt und der Zwischenspeicher damit mehr Werte pro Client speichern kann. Dieser Vorteil wirkt sich bei dynamischer Bindung nicht aus, da ein Replikat weiterhin von allen Clients angefragt werden kann.

Um die Auswirkung der Bindungsart auf Zwischenspeicher genauer untersuchen zu können, wurde eine Simulation durchgeführt, deren Ergebnisse in Abbildung 4.5 zu sehen sind. Das simulierte System besteht aus einem Objekt, das als Zwischenspeicher fungiert, und vier Clients. Die Clients fragen beim Objekt insgesamt 1000 Werte an, die das Objekt entweder aus seinem Zwischenspeicher entnimmt (Treffer) oder extern beschafft (Fehlschlag). Der Zwischenspeicher bietet Platz für 100 Werte. Bei einem Fehlschlag wird der entsprechende Wert in den Zwischenspeicher aufgenommen und verdrängt dabei den Wert, der am längsten nicht mehr angefragt wurde. Ein typisches Anfrageverhalten, wie es beispielsweise bei WWW-Benutzern vorkommen könnte wird wie folgt simuliert: Jeder Client verfügt über eine Menge von 50 bevorzugten Werten, die er mit einer Häufigkeit von 66% anfragt; die übrigen Anfragen entfallen gleichverteilt auf alle anderen Werte. Die Mengen der bevorzugten Werte der einzelnen Clients sind disjunkt. Die Abbildung zeigt die Trefferquote eines Replikats des Zwischenspeichers in Abhängigkeit von der Zeit und der Art der Bindung. Die einzelnen Clients werden nacheinander gestartet. Es ist deutlich zu sehen, wie die Trefferrate bei jedem zusätzlichen Client abfällt, da der Zwischenspeicher nun von mehreren Clients genutzt wird und dadurch weniger Werte pro Client speichern kann. Später werden dann drei Replikationen durchgeführt, so daß im Endzustand für jeden Client ein Replikat des Zwischenspeichers vorhanden ist. Im Fall der dynamischen Bindung wurden die Anfragen der Clients zyklisch auf die einzelnen Replikate verteilt, wobei die Trefferrate unverändert bleibt. Die statische Bindung hingegen bewirkt eine Erhöhung der Trefferrate, da sich nun sukzessive die Anzahl der Clients pro Replikat verringert und somit mehr Werte pro Client gespeichert werden können. Diese

Abbildung 4.5 Auswirkung der Bindungsart auf Zwischenspeicher



Simulation zeigt, daß die dynamische Bindung die Wirksamkeit der Lastverteilung erheblich einschränken kann.

Ein weiteres Problem bei der dynamischen Bindung ist die Lastverschiebung mit Zustandsübertragung. In Abschnitt 4.3.5 wurde beschrieben, daß die Zustandsübertragung die Kosten der Lastverschiebung bei Objekten mit redundantem Zustand stark verringern kann, da der Zustand nach der Migration oder Replikation nicht neu berechnet oder beschafft werden muß. Dieser Vorteil wirkt sich nur dann aus, wenn gewährleistet ist, daß die Clients der Quellreplikate an die entsprechenden Zielreplikate gebunden werden, wie es die statische Bindung fordert. Eine Bindung an andere Replikate könnte den Vorteil der Zustandsübertragung wieder zunichte machen.

Wegen der Probleme bei Objekten mit redundantem Zustand verzichtet dieses Lastverwaltungskonzept auf die dynamische Bindung. Dies hat den Nachteil, daß Clients nicht innerhalb einer Replikatgruppe umverteilt werden können. Da aber weiterhin die Möglichkeit besteht, neue Clients auf bestehende Replikate zu verteilen, sollte dieser Nachteil die Qualität der Lastverwaltung kaum beeinträchtigen.

4.4 Bewertung bestehender Lastverwaltungssysteme

Im folgenden werden einige Lastverwaltungssysteme vorgestellt und anhand des hier beschriebenen Konzepts zur Lastverwaltung in verteilten objektorientierten Systemen

bewertet. In Tabelle 4.2 sind die betreffenden Systeme im Überblick dargestellt. Besonderes Augenmerk wird dabei auf die Lastverteilung gerichtet, da diese Komponente die Grundlage für die Konzeption von Lastverwaltungssystemen bildet. Unterstützte Eigenschaften sind mit "+", nicht unterstützte mit "-" gekennzeichnet. Merkmale, die wegen einer fehlenden Basistechnologie nicht betrachtet werden können, sind durchgestrichen.

Tabelle 4.2 Ein Vergleich bestehender Lastverwaltungssysteme

		CLB	VisiBroker	Orbix	WINNER	TAO	dieses Konzept
Objekte	zustandslos	+	+	+	+	+	+
	zustandsbehaftet	−	−	−	−	−	+
	redundanter Zustand	−	−	−	−	−	+
Lastverteilung	Initialplazierung	+	−	−	−	−	+
	Migration	−	−	−	−	−	+
	Replikation	−	+	+	+	+	+
Bindung	statisch		−	(+)	(+)	−	+
	dynamisch		+	−	−	+	−
Transparenz	Lasterfassung	+			+	−	+
	Lastverteilung	+	−	−	−	−	+

Component Load Balancing (CLB) ist ein Lastverwaltungssystem für COM und DCOM basierte Anwendungen. Es ist Bestandteil des Application Center 2000 von Microsoft [29, 79], das Werkzeuge für den Betrieb und die Verwaltung von WWW basierten Anwendungen bereitstellt. Wie auch alle anderen Systeme, verwendet CLB Objekte als Lastverteilungseinheiten und Rechner bzw. Server-Prozesse als Ausführungseinheiten. CLB erweitert den Service Control Manager (SCM), der unter dem Betriebssystem Windows für die Erzeugung und Aktivierung von COM Objekten verantwortlich ist, um Funktionalität zur Lastverwaltung. Der SCM betreibt Initialplazierung, d.h er weist Objekten bei der Erzeugung einen geeigneten Rechner zu. COM Objekte werden über sogenannte Interface Identifier (IID) adressiert. Ein IID ist lediglich ein Identifikator für eine Schnittstelle, nicht für eine Objekt. Dies stellt eine wesentliche Einschränkung des Objektmodells hinsichtlich der Objektidentität dar, die sich auch auf die Lastverwaltung auswirkt: Die Lastverwaltung betrachtet alle Objekte mit gleicher Schnittstelle als identisch. Da dies nur für zustandslose Objekte gilt, ist auch die Lastverwaltung auf zustandslose Objekte beschränkt. CLB unterstützt auch die Wiederverwendung bestehender Objekte, die Entfernung nicht mehr genutzter Objekte (Garbage Collection) sowie die Reaktivierung bereits terminierter Objekte. Die Lasterfassung und die Initialplazierung sind transparent.

Die CORBA Implementierung VisiBroker [143] bietet Lastverwaltung durch sogenannte SmartAgents. Die SmartAgents bilden einen verteilten Verzeichnisdienst, bei dem alle aktiven Objekte registriert sind. Objekte gleichen Typs werden zu einer Replikatgruppe zusammengefaßt. Clients fragen bei einem SmartAgent ein Objekt eines bestimmten Typs an, woraufhin sie die Objektreferenz eines Replikats erhalten. Lastverteilung wird erreicht, indem verschiedene Clients an unterschiedliche Replikate gebunden werden. Die Zuordnung von Clients und Replikaten geschieht zyklisch, weshalb auch keine Lasterfassung nötig ist. Mit diesem Verfahren kann weder Lastausgleich noch Lastbalancierung erreicht werden, da die Leistungsfähigkeit und die Belastung der einzelnen Rechner nicht berücksichtigt wird. Der VisiBroker unterstützt dynamische Replikatgruppen, d.h. es können zur Laufzeit Replikate entfernt oder hinzugefügt werden. Der Broker bindet Clients, deren Replikat terminiert automatisch an ein anderes Element der Replikatgruppe. Die Bindung von Clients an Replikate ist dynamisch, da die Clients eines terminierten Replikats auf alle übrigen Elemente der Replikatgruppe verteilt werden. Die dynamische Bindung wirkt sich in diesem Fall jedoch nicht negativ auf die Wirksamkeit der Lastverteilung aus, da nur zustandslose Objekte betrachtet werden. Die Lastverteilung ist nicht-transparent, da das CORBA Objektmodell die Verwendung von SmartAgents nicht vorsieht.

Eine ähnliche Vorgehensweise findet man auch bei der Lastverwaltung der CORBA Implementierung Orbix [54, 55]. Statt der SmartAgents verwendet Orbix den CORBA Namensdienst und erweitert diesen um Funktionalität zur Verwaltung von Replikatgruppen. Unterschiedliche Clients werden entweder zyklisch oder zufallsverteilt an Replikate gebunden, wodurch auf die Lasterfassung verzichtet werden kann. Die Lastverwaltung beschränkt sich ebenfalls auf zustandslose Objekte. Die Bindung von Clients an Replikate ist statisch, d.h. es ist gewährleistet, daß ein Client stets auf dasselbe Replikat zugreift. Da Orbix zur Bindung von Clients denselben Mechanismus verwendet wie auch der VisiBroker, wird die statische Bindung nur dadurch sichergestellt, daß Clients deren Replikat terminiert nicht automatisch an ein anderes Replikat gebunden werden. Damit ist die Transparenz der Lastverteilung noch weiter eingeschränkt, als es schon beim VisiBroker der Fall ist.

Auch das Lastverwaltungssystem WINNER [4] erweitert den CORBA Namensdienst um Funktionalität zur Verwaltung von Replikatgruppen. Im Gegensatz zu Orbix verfügt WINNER über eine Lasterfassungskomponente, mit deren Hilfe das Replikat bzw. der Rechner mit der geringsten Last ermittelt werden kann. Die Lasterfassung ist auf Ebene des Ablaufsystems realisiert und somit transparent. Wie auch bei den anderen Systemen beschränkt sich die Lastverwaltung auf zustandslose Objekte. Die Bindung von Clients an Replikate kann auch bei WINNER nur eingeschränkt als statisch bezeichnet werden, da die Lastverwaltung beim Terminieren eines Replikats dessen Clients nicht automatisch an andere Replikate bindet. Die Lastverteilung ist, wie bei allen Systemen, die auf dem CORBA Namensdienst basieren nicht-transparent, da der CORBA Standard die Verwendung des Namensdienstes nicht zwingend vorschreibt.

Die Lastverwaltung der CORBA Implementierung TAO [115], die bereits in Kapitel 3.3.3 klassifiziert wurde, ist in Anlehnung an die CORBA Spezifikation zur Fehlertoleranz [92] entstanden. TAO unterstützt die Replikation zustandsloser Objekte. Die

Bindung von Clients an Replikate ist dynamisch, da bei der Replikation nicht gewährleistet ist, daß die Clients des Quellreplikats ausschließlich zwischen dem Quell- und dem Zielreplikat aufgeteilt werden. Die Verwaltung der Replikatgruppen orientiert sich stark an der Fehlertoleranz. Die Funktionalität zur Replikation muß der Entwickler zur Verfügung stellen, indem er vorgegebene Schnittstellen des Lastverwaltungssystems implementiert. Auch die Lasterfassung ist auf Anwendungsebene realisiert, d.h. die entsprechende Funktionalität muß vom Entwickler bereitgestellt werden. Somit sind Lasterfassung und Lastverteilung nicht-transparent.

Alle aufgeführten Lastverwaltungssysteme weisen eine Reihe von Mängeln auf. Ein Hauptkritikpunkt ist die mangelnde Unterstützung zustandsbehafteter Objekte. In der Literatur wird häufig argumentiert, daß diese Einschränkung dadurch umgangen werden kann, indem man den Objektzustand auslagert [79]. Das Auslagern des Zustandes, beispielsweise in eine Datenbank, kann jedoch die Kosten für Datenzugriffe signifikant erhöhen, da Zugriffe auf den Objektzustand dann über Prozeß- oder sogar Rechnergrenzen hinweg erfolgen. Dies ist wesentlich teurer als lokale Speicherzugriffe und führt bei Objekten mit großem Datenvolumen oder vielen Datenzugriffen zu massiven Performanzverlusten. Weiterhin muß berücksichtigt werden, daß die Auslagerung auf ein zentrales Medium die Skalierbarkeit der gesamten Anwendung stark beeinträchtigt. Beispielsweise kann eine zentrale Datenbank in einem großen System schnell zu einer Art Flaschenhals werden. Diese Nachteile umgehen viele Entwickler, indem sie Zwischenspeicher verwenden, um die Anzahl der Zugriffe auf ausgelagerte Daten zu minimieren. Diese Vorgehensweise führt jedoch, wie in Abschnitt 4.3.4 beschrieben, zu Objekten mit redundantem Zustand. Die Klasse der Objekte mit redundantem Zustand ist, analog zu zustandslosen Objekten, replizierbar, jedoch kann in diesem Fall die Wirksamkeit der Replikation stark eingeschränkt sein, wie in den Abschnitten 4.3.5 und 4.3.6 gezeigt wurde. Keines der untersuchten Lastverwaltungssysteme betrachtet Objekte mit redundantem Zustand bzw. die, bei der Replikation, möglichen Optimierungen. Unabhängig von dieser Diskussion gibt es jedoch auch eine Klasse von Objekten, bei welchen die Auslagerung des Zustandes nicht möglich ist, sei es aus Gründen der Performanz oder wegen konzeptioneller Gegebenheiten. Für diese Klasse von Objekten ist die Migration neben der Initialplazierung der einzig mögliche Lastverteilungsmechanismus. Keines der hier beschriebenen Lastverwaltungssysteme verfügt über die Möglichkeit zur Migration. Dies liegt daran, daß eine Migration bei zustandslosen Objekten durch eine Replikation und das anschließende Entfernen eines Replikats nachgebildet werden kann. Bei zustandsbehafteten Objekten ist dies jedoch nicht möglich, da sie nicht replizierbar sind. Generell stellt die Beschränkung auf zustandslose Objekte eine starke Einschränkung für den Entwickler dar, was zu mangelnder Akzeptanz der Lastverwaltung führt.

Die Bindung von Clients an Replikate ist bei den meisten untersuchten Lastverwaltungssystemen dynamisch. Orbix und WINNER unterstützen zwar die statische Bindung, können diese aber nur dank ihrer sehr eingeschränkten Möglichkeiten zur Lastverteilung gewährleisten. Die Beschränkung auf die dynamische Bindung wirkt sich bei den vorliegenden Systemen jedoch nicht negativ auf die Wirksamkeit der Lastverteilung aus, da nur zustandslose Objekte betrachtet werden.

Ein weiterer Kritikpunkt ist die mangelnde Transparenz und die schlechte Integration in das zugrundeliegende Objektmodell. Die Realisierung der Lastverwaltung über Namens- oder Verzeichnisdienste, wie es bei VisiBroker, Orbix und WINNER der Fall ist, stellt einen Bruch mit dem CORBA Objektmodell dar, da dieses die Verwendung eines Namensdienstes nicht zwingend vorschreibt. Das Lastverwaltungssystem TAO hingegen realisiert die Lastverwaltung über einen externen Dienst. Damit muß die Integration der Lasterfassung und der Lastverteilung auf Anwendungsebene erfolgen, was jedoch per Definition nicht-transparent ist. Dies ist mit einem signifikanten Mehraufwand für den Entwickler verbunden, was die Akzeptanz der Lastverwaltung einschränkt.

Das hier vorgestellte Konzept zur Lastverwaltung in verteilten objektorientierten Systemen beseitigt die beschriebenen Mängel. Einer der Kernpunkte dieses Konzepts ist die Unterstützung von zustandslosen und zustandsbehafteten Objekten, um damit ein möglichst breites Spektrum an Anwendungen bedienen zu können. Des weiteren setzt dieses Konzept auf die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen, wie der Initialplazierung, der Migration und der Replikation. Von der kombinierten Anwendung unterschiedlicher Lastverteilungsmechanismen sind signifikante Synergieeffekte für die Lastverwaltung zu erwarten, da sich sowohl das Spektrum der unterstützten Anwendungen als auch die Einflußmöglichkeiten der Lastverwaltung vergrößern. Besonders Augenmerk wird bei diesem Konzept auf die Replikation gerichtet. Durch die Einführung des redundanten Zustandes ist es möglich, für diese Klasse von Objekten spezielle Optimierungen wie die Zustandsübertragung und die statische Bindung anzubieten. Damit kann die Wirksamkeit der Lastverteilung stark erhöht werden. Ein weiterer Kernpunkt dieses Konzepts ist die Transparenz der Lastverwaltung. Sowohl die Lasterfassung als auch die Lastverteilung sollten möglichst transparent gestaltet sein, um den Mehraufwand für den Entwickler so gering als möglich zu halten und damit die Akzeptanz der Lastverwaltung zu steigern.

4.5 Zusammenfassung

In diesem Kapitel wurden zunächst die wichtigsten Charakteristika verteilter objektorientierter Systeme beschrieben und daraus die Anforderungen an die Lastverwaltung abgeleitet. Aus diesen Anforderungen entstand ein Lastverwaltungskonzept, dessen Kernpunkte die Kombination unterschiedlicher Lastverteilungsmechanismen und die Transparenz sind. Die kombinierte Anwendung verschiedener Lastverteilungsmechanismen ermöglicht es der Lastverwaltung, den unterschiedlichen Ursachen für die Entstehung von Lastungleichheit entgegenzuwirken. Mit Hilfe der Migration kann Hintergrundlast kompensiert werden, wohingegen die Replikation geeignet ist, hohe Anfragelasten auf mehrere Rechner zu verteilen. Die Transparenz der Lastverwaltung und ihre nahtlose Integration in das zugrundeliegende Programmiermodell tragen entscheidend dazu bei, die Akzeptanz der Lastverwaltung beim Anwender zu steigern. Ein weiterer Kernpunkt dieses Lastverwaltungskonzepts ist die Unterstützung zustandsloser und zustandsbehafteter Objekte. Die ausschließliche Betrachtung zustandsloser Objekte würde die Realisierung des Konzepts erleichtern, da keine Per-

sistenzmechanismen nötig sind. Dies hätte jedoch eine erhebliche Einschränkung des Handlungsspielraums der Lastverwaltung zur Folge, wodurch Lastausgleich und Lastbalancierung wesentlich schwerer zu erreichen wären.

Im weiteren Verlauf des Kapitels wurden Objekte und ihre speziellen Eigenschaften wie die Identität, die Äquivalenz und der Zustand näher untersucht, um Aussagen über die Migrierbarkeit und insbesondere die Replizierbarkeit treffen zu können. Es hat sich gezeigt, daß der Objektzustand kein geeignetes Kriterium zur Bestimmung der Objektäquivalenz und damit auch der Replizierbarkeit ist. Aus diesem Grund wurden aus den Ablaufprotokollen der Migration und der Replikation Kriterien für die Migrierbarkeit und die Replizierbarkeit von Objekten abgeleitet. Das Ziel dieser Vorgehensweise war es, dem Entwickler eine Reihe von Entscheidungskriterien an die Hand zu geben, mit denen er feststellen kann, ob ein Objekt migrierbar bzw. replizierbar ist. In diesem Zusammenhang wurde der Begriff des redundanten Zustandes eingeführt. Die Definition des redundanten Zustandes stützt sich auf der Objektäquivalenz ab und berücksichtigt dadurch auch das Verhalten von Objekten. Mit Hilfe des redundanten Zustandes ist es nun möglich, Aussagen über die Replizierbarkeit von Objekten zu treffen. Anschließend wurde die Lastverteilung bei Objekten mit redundantem Zustand näher untersucht. Dabei hat sich gezeigt, daß diese Klasse von Objekten gesondert betrachtet werden muß, da es Optimierungen gibt, welche die Wirksamkeit der Lastverteilung enorm steigern.

Zum Abschluß des Kapitels wurden einige Lastverwaltungssysteme vorgestellt und mit dem hier präsentierten Konzept verglichen. Keines der existierenden Systeme erfüllt alle, zu Beginn des Kapitels gestellten Anforderungen. Dies war schließlich die Motivation für ein neues Konzept zur Lastverwaltung in verteilten objektorientierten Systemen, das in diesem Kapitel beschrieben wurde.

5. Ein Lastmodell für verteilte objektorientierte Systeme

In Kapitel 3 wurde ein Klassifikationsmodell für Lastverwaltungssysteme vorgestellt. Der Schwerpunkt dieses Modells liegt auf der Beschreibung der grundlegenden Eigenschaften und der Funktionsweise von Lastverwaltungssystemen. Konkrete Strategien und Algorithmen zur Lastbewertung sind darin nicht enthalten, da diese eine wesentlich detaillierte Betrachtungsweise voraussetzen, als es auf dem Abstraktionsniveau eines Klassifikationsschemas möglich ist. Zu diesem Zweck wird hier der Begriff des Lastmodells eingeführt. Ein Lastmodell beschreibt die einzelnen Kenngrößen, die von einem Lastbewertungsalgorithmus herangezogen werden, um Entscheidungen über die Lastverteilung zu treffen. Mit Hilfe eines Lastmodells können dann Algorithmen zur Lastbewertung formuliert und beschrieben werden.

Ziel dieses Kapitels ist es, ein Lastmodell für verteilte objektorientierte Systeme zu entwickeln. Anschließend wird anhand des Lastmodells ein Algorithmus zur Lastbewertung formuliert, der dann im nachfolgenden Kapitel als Grundlage für die Implementierung der Lastbewertungskomponente dient.

5.1 Das Lastmodell

Der Begriff des Lastmodells geht auf LUDWIG [71] zurück. In Anlehnung daran sei ein Lastmodell wie folgt definiert:

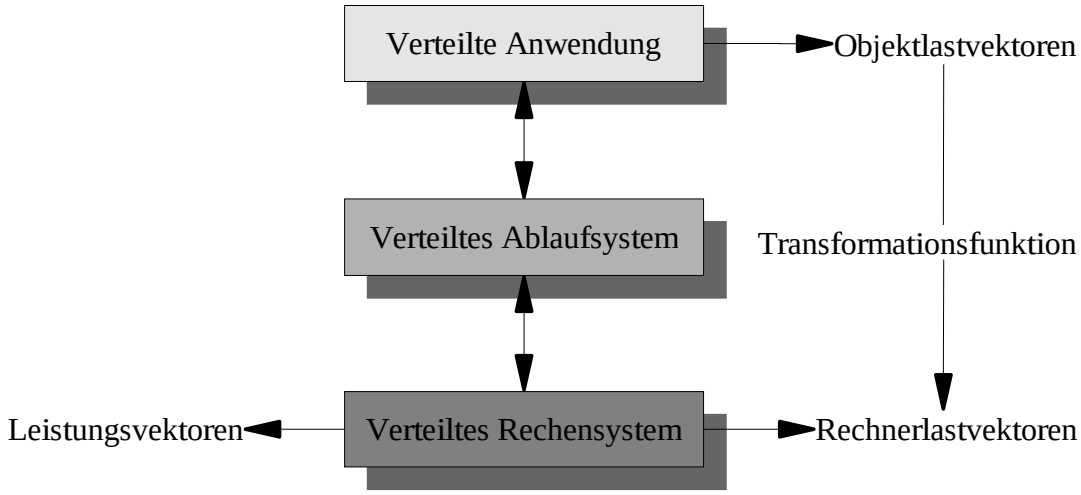
Definition 5.1 *Lastmodell*

> Ein Lastmodell ist ein n-Tupel von Kenngrößen, die von einem Lastbewertungsalgorithmus zur Entscheidungsfindung herangezogen werden. □

Die betrachteten Kenngrößen beinhalten sowohl die Lastinformation als auch die Art ihrer Verknüpfung.

Wegen der Schichtung verteilter objektorientierter Systeme muß ein geeignetes Lastmodell mehrere Ebenen eines verteilten Systems abdecken. Abbildung 5.1 zeigt die grundlegenden Kenngrößen eines solchen Lastmodells. Auf der Ebene des Rechensy-

Abbildung 5.1 Ein mehrschichtiges Lastmodell für verteilte objektorientierte Systeme



stems findet man zwei Arten von Lastinformation: Leistungsvektoren beschreiben die Leistungsfähigkeit der Rechner bzw. ihrer Ressourcen. Die Auslastung der Rechner wird durch Rechnerlastvektoren ausgedrückt. Auf Anwendungsebene liefert das Lastmodell Information über Objekte und deren Auslastung, die in den Objektlastvektoren zusammengefaßt ist. Da sich die Lastinformation der Anwendung stark von der Lastinformation des Rechensystems unterscheidet, benötigt man eine Kenngröße, um die Lastinformation der unterschiedlichen Ebenen vergleichbar zu machen. Diese Aufgabe erfüllt die Transformationsfunktion, mit welcher Objekt- auf Rechnerlastvektoren abgebildet werden. Formal stellt sich dieses Lastmodell wie folgt dar:

Definition 5.2 *Ein Lastmodell für verteilte objektorientierte Systeme*

Ein Lastmodell für verteilte objektorientierte Systeme ist ein Vier-Tupel $\mathcal{M} = (\mathcal{L}, \mathcal{R}, \mathcal{O}, t)$ bestehend aus:

1. Einer Menge von Leistungsvektoren:
   $\mathcal{L} = \{l_1, \ldots, l_m\}$ mit l_λ als Leistungsvektor des Rechenrs λ
   und $\forall \lambda = 1, \ldots, m : l_\lambda \in (\mathbb{R}^+)^i$ mit $m \geq 1, \quad i \geq 1$
2. Einer Menge von Rechnerlastvektoren:
   $\mathcal{R} = \{r_1, \ldots, r_m\}$ mit r_λ als Rechnerlastvektor des Rechenrs λ
   und $\forall \lambda = 1, \ldots, m : r_\lambda \in (\mathbb{R}_0^+)^i$ mit $m \geq 1, \quad i \geq 1$
3. Einer Menge von Objektlastvektoren:
   $\mathcal{O} = \{o_1, \ldots, o_n\}$ mit o_λ als Objektlastvektor des Objekts λ
   und $\forall \lambda = 1, \ldots, n : o_\lambda \in (\mathbb{R})^j$ mit $n \geq 1, \quad j \geq 1$
4. Der Transformationsfunktion:
   $t : \mathcal{O} \times \mathcal{L} \rightarrow \mathcal{R}$

□

Dieses Lastmodell für verteilte objektorientierte Systeme ist nur eine Möglichkeit, die einzelnen Kenngrößen der Lastbewertung darzustellen; verschiedene Variationen sind denkbar. Der hier gewählte Ansatz orientiert sich stark an den Gegebenheiten real existierender Systeme. Dies gewährleistet einen engen Bezug des theoretischen Modells zu seiner Realisierung. Im folgenden werden die einzelnen Kenngrößen dieses Lastmodells vorgestellt.

Die erste Kenngröße auf der Ebene des Rechensystems sind die Leistungsvektoren. Sie beschreiben die maximale Leistungsfähigkeit des Rechensystems und seiner Ressourcen. Jeder Rechner verfügt über einen Leistungsvektor, dessen Elemente wiederum die maximale Kapazität der Ressourcen des Rechners darstellen. Die Zuordnung der Ressourcen und der einzelnen Elemente eines Leistungsvektors muß bei allen Rechnern identisch sein, um ihre Vergleichbarkeit zu gewährleisten. Als ein Beispiel für ein Element eines Rechnerlastvektors sei hier die Netzwerklast genannt. Die Netzwerklast sei entsprechend der Definition 3.2 als das Verhältnis der nachgefragten und der maximal verfügbaren Übertragungskapazität des Netzwerks definiert.

Eine weitere Kenngröße des Rechensystems sind die Rechnerlastvektoren. Analog zu den Leistungsvektoren ist jedem Rechner ein Rechnerlastvektor zugeordnet. Jedes Element eines Rechnerlastvektors entspricht der Last einer Ressource des betreffenden Rechners. Die Zuordnung von Ressourcen und Vektorelementen ist dabei dieselbe, wie bei den Leistungsvektoren. Aufgrund der Heterogenität verteilter Systeme sind die Lasten unterschiedlicher Rechner nicht unbedingt vergleichbar, da die Last nach Definition 3.2 sowohl von der nachgefragten als auch von der maximal verfügbaren Kapazität abhängt. Die maximal verfügbare Kapazität vergleichbarer Ressourcen kann jedoch wegen der Heterogenität des Rechensystems stark variieren. Um die Lasten heterogener Rechner dennoch vergleichen zu können, wird mit Hilfe der Leistungsvektoren die Ressourcennachfrage eines Rechners berechnet. Die Ressourcennachfrage ist unabhängig von der Beschaffenheit des Rechensystems und damit auf jeden beliebigen Vergleichsrechner übertragbar. Mit Hilfe eines entsprechenden Leistungsvektors kann daraus wieder die Last berechnet werden. Dieser Sachverhalt wird im folgenden anhand des oben genannten Beispiels der Netzwerklast verdeutlicht. Die maximale Übertragungskapazität unterschiedlicher Netzsegmente kann stark variieren. Exemplarisch werden hier zwei Netze mit einer maximalen Übertragungskapazität von 10 MBit/s bzw. 100 MBit/s betrachtet, die in den Leistungsvektoren enthalten ist. Das Netz mit 10 MBit/s wird von genau einem Verbraucher genutzt, der eine Netzwerklast von $0{,}2$ verursacht. Die Frage lautet nun: Welche Last würde dieser Verbraucher an einem Netzwerk mit einer Übertragungskapazität von 100 MBit/s hervorrufen? Zur Klärung dieser Frage wird zuerst die Ressourcennachfrage des Verbrauchers berechnet. Nach Gleichung 3.2 ergibt sich aus der Last und der maximalen Kapazität des Netzwerks, daß der Verbraucher eine Bandbreite von 2 MBit/s für sich beansprucht. Die Ressourcennachfrage wird nun auf ein Netzwerk mit 100 MBit/s übertragen. Nach Gleichung 3.2 ergibt sich aus der Ressourcennachfrage und der maximalen Kapazität des Netzwerks, daß der Verbraucher dort eine eine Last von $0{,}02$ verursachen würde. Dieses Beispiel veranschaulicht, wie mit Hilfe der Leistungsvektoren die Lasten heterogener Ressourcen verglichen werden können.

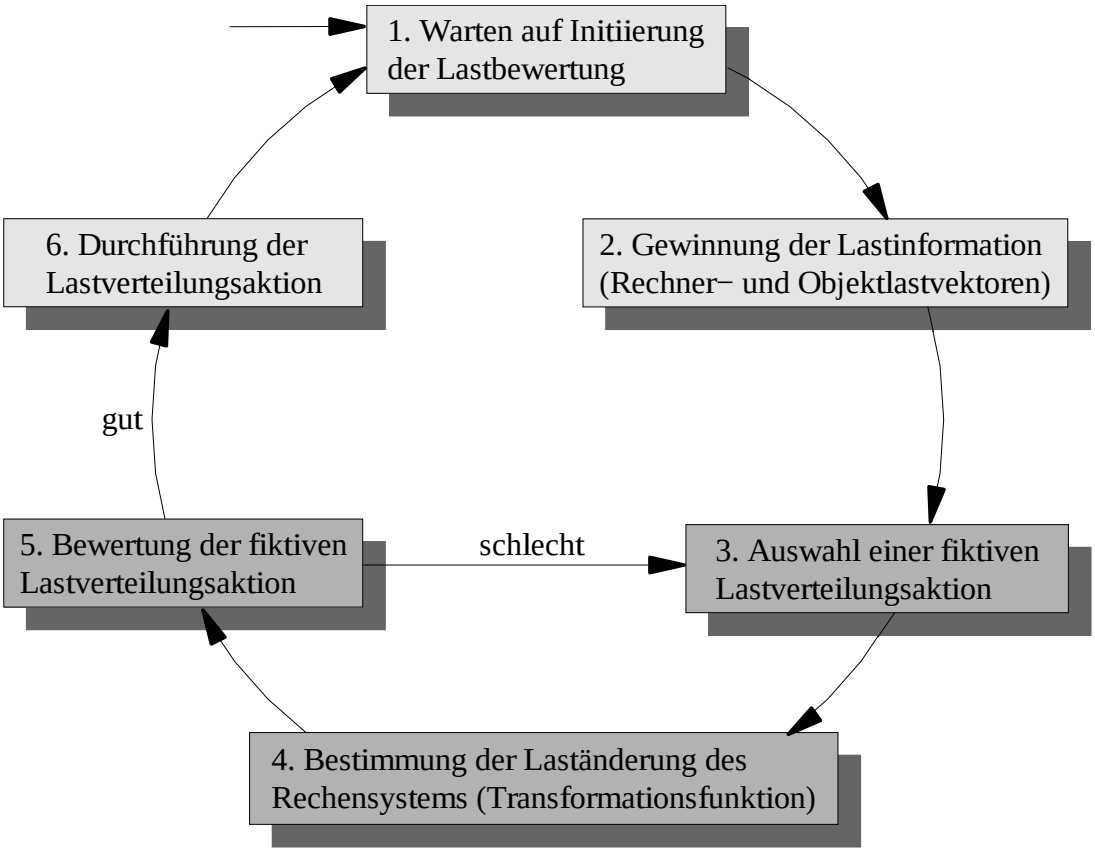
Die Ressourcen der Anwendungsebene sind Objekte. Insofern sind Objektlastvektoren die bestimmende Kenngröße auf der Ebene der Anwendung. Im Gegensatz zu einem Rechnerlastvektor beziehen sich alle Elemente eines Objektlastvektors auf ein Objekt, d.h. auf eine Ressource. Die einzelnen Elemente der Objektlastvektoren sind reelle Zahlen, die in ihrer Gesamtheit die Arbeitslast eines Objekts darstellen. Betrachtet man beispielsweise die Netzwerklast, die ein Objekt erzeugt, so stellt man fest, daß diese Last nicht direkt ermittelt werden kann. Auf der Ebene der Anwendung sind lediglich einzelne Methodenaufrufe zu erkennen, deren Parameter über das Netzwerk übertragen werden. Die daraus resultierende Netzwerklast ist sowohl durch die Beschaffenheit der Parameter als auch die Häufigkeit der Methodenaufrufe bestimmt. Die einzelnen Elemente eines Objektlastvektors haben also keinen direkten Bezug zu den Ressourcen des Rechensystems. Deshalb ist es für einen Lastbewertungsalgorithmus unmöglich, die Information eines Objektlastvektors zu interpretieren.

Um dennoch eine Beziehung zwischen der Lastinformation des Rechensystems und der Anwendung herstellen zu können, verfügt das Lastmodell über eine Transformationsfunktion. Die Transformationsfunktion beinhaltet ein stark vereinfachtes Modell des verteilten Ablaufsystems und ermöglicht damit die Abbildung von Objekt- auf Rechnerlastvektoren. Damit kann ermittelt werden, wie stark ein Objekt die Ressourcen seines ausführenden Rechners belastet. Auf diese Art und Weise kann die Lastinformation über die Anwendung so aufbereitet werden, daß sie von einem Lastbewertungsalgorithmus interpretiert werden kann. Auch dieser Sachverhalt sei anhand der Netzwerklast verdeutlicht: Wie bereits beschrieben, enthalten die Objektlastvektoren Information über die Größe der Parameter und die Aufrufhäufigkeit der Methoden eines Objekts. Die Transformationsfunktion kann mit Hilfe ihres Wissens über das verteilte Ablaufsystem aus diesen Einzelwerten die Netzwerklast eines Objekts bestimmen. Diese kann wiederum von einem Lastbewertungsalgorithmus interpretiert werden, da sie in direkter Beziehung zur Lastinformation des Rechensystems steht.

5.2 Ein Vorgehensmodell

Das hier vorgestellte Lastmodell für verteilte objektorientierte Systeme umfaßt eine Menge von Kenngrößen, die ein Lastbewertungsalgorithmus zur Entscheidungsfindung heranziehen kann. Ein Lastmodell allein sagt noch nichts über den Algorithmus oder die Strategie zur Lastbewertung aus. Auf der Basis eines Lastmodells können jedoch konkrete Lastbewertungsalgorithmen beschrieben und formuliert werden. In diesem Zusammenhang spricht man auch von der Realisierung des Lastmodells. Das hier beschriebene Modell ist sehr allgemein gehalten, so daß damit eine Vielzahl unterschiedlicher Strategien realisierbar ist. Um die prinzipielle Funktionsweise konkreter Lastbewertungsstrategien aufzuzeigen, wird ein Vorgehensmodell für dieses Lastmodell vorgestellt. Abbildung 5.2 zeigt das Vorgehensmodell im Überblick. Es umfaßt insgesamt sechs Schritte, die in zwei Kreisläufen angeordnet sind. Bei einem Durchlauf des äußeren Kreislaufs kann der innere, dunkel dargestellte Kreislauf mehrfach durchlaufen werden. Im folgenden sind die einzelnen Schritte aufgeführt:

Abbildung 5.2 Ein Vorgehensmodell für das Lastmodell



1. Warten auf Initiierung der Lastbewertung: Die Initiierung der Lastbewertung ist der Ausgangspunkt für die Betrachtung dieses Vorgehensmodells. Die Steuerung und die Initiierung der Lastbewertung kann, wie in Kapitel 3.2.2 erläutert wurde, last- oder zeitgesteuert erfolgen. Beide Varianten haben sowohl Vor- als auch Nachteile. Die Entscheidung für eine der beiden Möglichkeiten ist eine Frage des Designs der Lastbewertungskomponente.

2. Gewinnung der Lastinformation: Die Lastbewertung erhält von der Lasterfassungskomponente zur Laufzeit Lastinformation über die Ressourcen des Rechensystems und der Anwendung. Die Lastinformation über das Rechensystem ist in den Rechnerlastvektoren enthalten. Die Last der Anwendung bzw. ihrer Objekte beschreiben die Objektlastvektoren. Mit Hilfe der Leistungsvektoren können die Lasten heterogener Ressourcen vergleichbar gemacht werden.

3. Auswahl einer fiktiven Lastverteilungsaktion: Kernpunkt dieses Vorgehensmodells ist die Durchführung fiktiver Lastverteilungsaktionen. Dabei wird anhand einer noch näher zu spezifizierenden Strategie eine mögliche Lastverteilungsaktion ausgewählt.

4. Bestimmung der Laständerung des Rechensystems: Die Durchführung einer Lastverteilungsaktion würde die Lasten der betroffenen Rechner verändern. Die Lastverteilungsstrategie beinhaltet ein Modell des verteilten Systems, anhand dessen sie die Laständerung des Rechensystems ermittelt. Grundlage dafür ist die Transformationsfunktion, mit der bestimmt werden kann, wie stark ein Objekt die Ressourcen seines ausführenden Rechners belastet.

5. Bewertung der fiktiven Lastverteilungsaktion: Die fiktive Lastverteilungsaktion kann nun anhand der veränderten Lastverhältnisse bewertet werden. Grundlagen dafür sind das Ziel der Lastverwaltung sowie die gewählten Randbedingungen. Wird eine Aktion als tauglich befunden, so kann mit dem nächsten Schritt fortgefahren werden, anderenfalls ist eine andere Aktion auszuwählen.

6. Durchführung der Lastverteilungsaktion: Falls eine Lastverteilungsaktion gefunden wurde, die den Zielen und Randbedingungen der Lastverwaltung entspricht, kann diese mit Hilfe der Lastverteilungskomponente ausgeführt werden.

Das Vorgehensmodell weist starke Parallelen zur Regelungstechnik auf. Moderne Regler verfügen über ein mathematisches Modell des zu regelnden Systems. Anhand des Modells kann der Regler die Auswirkungen eines Eingriffs in das System abschätzen, bevor diese wirksam werden und über die Rückkopplung am Meßglied anliegen. Diese Vorgehensweise ist insbesondere dann vorteilhaft, wenn ein Eingriff in das System erst mit großer Zeitverzögerung meßbar ist. Dies ist auch bei verteilten objektorientierten Systemen der Fall, was wiederum für diesen Ansatz spricht.

5.3 Die Lastbewertungsstrategie

Die folgenden Abschnitte dieses Kapitels sind der Realisierung des Lastmodells gewidmet. Dabei wird zuerst auf die Lastbewertungsstrategie eingegangen. Die Realisierung der einzelnen Kenngrößen des Lastmodells folgt erst am Schluß, um zu verhindern, daß die Strategie von implementierungstechnischen Details beeinflußt wird.

5.3.1 Ziel der Lastbewertungsstrategie

Das Ziel der hier vorgestellten Lastbewertungsstrategie ist die Lastbalancierung. Bei der Lastbalancierung wird gefordert, daß sich die Lasten vergleichbarer Ressourcen höchstens um ein quantitatives Maß unterscheiden. Diese Problemstellung ist NP-vollständig, wie GAREY und JOHNSON [35] bereits im Jahr 1979 zeigten [31, 71, 11]. Dies bedeutet, daß die Lastbalancierung im allgemeinen Fall nur in exponentieller Zeit lösbar ist. Konkrete Lösungsansätze versuchen deshalb eine Näherung in polynomialer Zeit zu berechnen. Zu diesem Zweck wird hier, in Anlehnung an das Vorgehensmodell aus Abschnitt 5.2, versucht die Lastbalancierung schrittweise durch die Ausführung von Lastverteilungsaktionen anzunähern.

Zur Bewertung aller möglichen Lastverteilungsaktionen benötigt man ein Maß, um das Lastungleichgewicht des Rechensystems zu beschreiben. Damit können dann die einzelnen Lastverteilungsaktionen hinsichtlich ihrer Wirksamkeit bewertet werden. Das Lastmodell aus Abschnitt 5.1 betrachtet jedoch mehrere Ressourcen, was wiederum die Frage aufwirft, wie Lastunterschiede zwischen verschiedenartigen Ressourcen zu bewerten sind. Durch die begrenzte Granularität der Lastverteilung kann das Problem auftreten, daß eine Lastverteilungsaktion ein Lastungleichgewicht zwischen den Ressourcen eines Typs verringert, dabei aber das Lastungleichgewicht eines anderen Ressourcen-Typs vergrößert. Um diesen Konflikt zu lösen, benötigt man ein ganzheitliches Maß für die Bewertung des Lastungleichgewichts des Rechensystems. Dazu wird die Bewertungszahl B wie folgt definiert:

$$B = \frac{1}{i \cdot m} \cdot \sum_{\mu=1}^{i} \sum_{\lambda=1}^{m} \left(r_{\lambda_\mu} - \frac{1}{m} \cdot \sum_{\xi=1}^{m} r_{\xi_\mu} \right)^2 \tag{5.1}$$

Die Bewertungszahl beschreibt zunächst das Lastungleichgewicht vergleichbarer Ressourcen anhand der Varianz ihrer Lasten. Das Lastungleichgewicht des gesamten Rechensystems ergibt sich dann aus dem arithmetischen Mittel der Lastungleichgewichte der einzelnen Ressourcen-Typen; jeder Ressourcen-Typ geht also gleichgewichtet in die Bewertungszahl ein. Folglich darf eine Lastverteilungsaktion, die das Lastungleichgewichts zwischen den Ressourcen eines Typs verringert, das Lastungleichgewicht bei den übrigen Ressourcen-Typen höchstens in gleichem Maße erhöhen. Das Ziel dieser Lastbewertungsstrategie ist es, die Bewertungszahl B und damit das Lastungleichgewicht des Rechensystems schrittweise durch Ausführung von Lastverteilungsaktionen zu minimieren.

5.3.2 Strategie zur Lastverschiebung

Als Mechanismen zur Lastverschiebung stehen, wie in Kapitel 4 beschrieben, die Migration und die Replikation von Objekten zur Verfügung. Dabei wird Last von einem Quellrechner q zu einem Zielrechner z verschoben; die Lasten der übrigen Rechner verändern sich nicht. Die Laständerungen des Quell- und des Zielrechners können wie folgt beschrieben werden:

$$\begin{aligned} r_q{}' &= (r_{q_1}{}', \ldots, r_{q_i}{}') \quad \text{mit} \quad \forall \mu = 1, \ldots, i : r_{q_\mu}{}' = r_{q_\mu} - v \cdot t(o, l_q)_\mu \\ r_z{}' &= (r_{z_1}{}', \ldots, r_{z_i}{}') \quad \text{mit} \quad \forall \mu = 1, \ldots, i : r_{z_\mu}{}' = r_{z_\mu} + v \cdot t(o, l_q)_\mu \cdot \frac{l_{q_\mu}}{l_{z_\mu}} \end{aligned} \tag{5.2}$$

Dabei ist der Anteil der Last des Objekts o, der auf den Zielrechner verschoben wird, durch den Verteilungswert v mit $0 < v \leq 1$ gekennzeichnet. Im Falle einer Migration ist v stets 1, da das gesamte Objekt und seine Arbeitslast auf den Zielrechner verschoben wird. Bei einer Replikation werden die Anfragen des ursprünglichen Objekts und damit auch die Last im Verhältnis $(1 - v) : v$ zwischen dem Replikat des Quell- und des Zielrechners aufgeteilt. Die Transformationsfunktion liefert die Last, die ein Objekt bei den Ressourcen des Quellrechners erzeugt. Diese Last kann, wie in Abschnitt

5.1 dargestellt ist, mit Hilfe der Leistungsvektoren auf den Zielrechner übertragen werden. Durch die Gewichtung mit dem Verteilungswert ergeben sich daraus die Lasten des Quell- und des Zielrechners nach Ausführung einer Lastverschiebeaktion.

Eine Lastverschiebeaktion wirkt sich auf das Lastungleichgewicht des Rechensystems und somit auch auf den Wert der Bewertungszahl aus. Die Bewertungszahl nach Ausführung einer Lastverschiebeaktion ergibt sich durch Einsetzen der Werte ${r_q}'$ und ${r_z}'$ in Gleichung 5.1:

$$\begin{aligned} B' =& \frac{1}{i \cdot m} \cdot \sum_{\mu=1}^{i} \left(\left(r_{z\,\mu} + v \cdot t(o,l_q)_\mu \cdot \frac{l_{q\,\mu}}{l_{z\,\mu}} - \overline{r_\mu} \right)^2 + \right. \\ & \left. \left(r_{q\,\mu} - v \cdot t(o,l_q)_\mu - \overline{r_\mu} \right)^2 + \sum_{\substack{\lambda=1 \\ \lambda \neq q, \lambda \neq z}}^{m} \left(r_{\lambda\,\mu} - \overline{r_\mu} \right)^2 \right) \\ & \text{mit} \quad \overline{r_\mu} = \frac{1}{m} \cdot \left(- v \cdot t(o,l_q)_\mu \cdot \left(1 - \frac{l_{q\,\mu}}{l_{z\,\mu}} \right) + \sum_{\xi=1}^{m} r_{\xi\,\mu} \right) \end{aligned} \tag{5.3}$$

Die Güte einer Lastverschiebeaktion kann nun bestimmt werden, indem man den Wert der Bewertungszahl vor (B) und nach der Ausführung der Aktion (B') vergleicht:

$$\epsilon = B - B' \tag{5.4}$$

Je größer ϵ ist, desto besser gleicht die Aktion das Lastungleichgewicht des Rechensystems aus.

Wie bereits erwähnt wurde, ist der Verteilungswert v bei einer Replikation so zu wählen, daß das Lastungleichgewicht des Rechensystems möglichst gering wird. Diese Aufgabenstellung entspricht der Minimierung der Bewertungszahl B' in Abhängigkeit von v. Zu diesem Zweck bildet man die Ableitung von B' nach v und bestimmt deren Nullstellen.[1] Damit ergibt sich als optimaler Verteilungswert:

$$\begin{aligned} v = & \sum_{\mu=1}^{i} \left(t(o,l_q)_\mu \cdot \left(r_{z\,\mu} \cdot \frac{l_{q\,\mu}}{l_{z\,\mu}} - r_{q\,\mu} + \frac{1}{m} \cdot \left(1 - \frac{l_{q\,\mu}}{l_{z\,\mu}} \right) \cdot \sum_{\lambda=1}^{m} r_{\lambda\,\mu} \right) \right) \quad / \\ & \sum_{\mu=1}^{i} \left({t(o,l_q)_\mu}^2 \cdot \left(\frac{1}{m} \cdot \left(1 - \frac{l_{q\,\mu}}{l_{z\,\mu}} \right)^2 - \left(\frac{l_{q\,\mu}}{l_{z\,\mu}} \right)^2 - 1 \right) \right) \end{aligned} \tag{5.5}$$

Dieser Verteilungswert ist ein globales Minimum der Bewertungszahl, da die zweite Ableitung von B' in jeder Umgebung von v größer als Null ist. Der Wertebereich von v ist auf das offene Intervall $]0,1[$ beschränkt. Ein Verteilungswert von kleiner oder gleich Null entstünde, wenn man zur Minimierung der Bewertungszahl Last vom Zielauf den Quellrechner verschieben müßte. Dies widerspricht jedoch der Definition der Lastverschiebung. Ein Verteilungswert von größer oder gleich Eins deutet darauf hin, daß eine Migration besser geeignet wäre.

[1] Die Bestimmung der Ableitungen und deren Nullstellen ist leicht nachvollziehbar, wird hier jedoch wegen des großen Umfangs der dabei entstehenden Terme nicht dargestellt.

Die hier beschriebene Strategie zur Lastverschiebung versucht, das Ziel der Lastbalancierung, wie es in Gleichung 5.1 formuliert ist, möglichst gut anzunähern. Vollständige Lastbalancierung ist in der Praxis schon wegen der begrenzten Granularität der Lastverteilung nicht realisierbar. Neben der streng mathematischen Zielsetzung der Lastbalancierung muß auch berücksichtigt werden, daß es sich bei der Lastverwaltung um eine planerische Aufgabe handelt. Die Lastbewertung betrachtet den Zustand des Systems anhand seiner aktuellen und historischen Lastwerte und versucht daraus Aktionen abzuleiten, welche die Lastsituation in der Zukunft verbessern. Dies erfordert eine vorausschauende Vorgehensweise. Schließlich kann eine Aktion, die im Moment sinnvoll erscheint, Ressourcen belegen, die eine zukünftige Aktion ebenfalls benötigen würde. Ein sparsames und planerisches Vorgehen bei der Ressourcenvergabe ist folglich eine wesentliche Randbedingung der Lastverwaltung. Zu diesem Zweck werden einige Anforderungen formuliert, die geeignete Lastverschiebeaktionen erfüllen müssen:

- Jede Lastverschiebeaktionen sollte einen gewissen Mindestbeitrag zur Lastbalancierung leisten. Dies verhindert, daß Ressourcen belegt werden, die man zu einem späteren Zeitpunkt wesentlich gewinnbringender nutzen könnte. Darüber hinaus trägt diese Forderung dazu bei, Aktionen zu vermeiden, deren Nutzen so gering ist, daß er die Kosten nicht aufwiegt.

- Es sollten nur Objekte repliziert werden, die über eine sehr hohe Anfragelast verfügen, um eine zu starke Fragmentierung der Anwendung zu verhindern. Zudem ist es bei Objekten mit großer Arbeitslast einfacher, die gewünschte Verteilung der Anfragen auf die einzelnen Replikate zu erreichen, da man davon ausgehen kann, daß hinreichend viele Anfragen zur Verfügung stehen.

Um Lastverschiebeaktionen zu vermeiden, deren Beitrag zur Lastbalancierung zu gering ist, wird der Schwellenwert Δ_ϵ eingeführt. In Anlehnung an Gleichung 5.4 muß für alle Lastverteilungsaktionen gelten:

$$\epsilon \geq \Delta_\epsilon \quad \text{mit} \quad \Delta_\epsilon \in \mathbb{R}^+ \tag{5.6}$$

Damit berücksichtigt die Lastbewertung nur Aktionen, welche die Bewertungszahl und damit das Lastungleichgewicht des Rechensystems um mindestens Δ_ϵ verringern.

Zur Umsetzung der zweiten Forderung muß bei der Replikation verlangt werden, daß die Last des betroffenen Objekts größer ist, als ein Schwellenwert Δ_R. In Anlehnung an Gleichung 5.2 betrachtet die Lastbewertung nur Replikationen, für die gilt:

$$\frac{1}{i} \cdot \sum_{\mu=1}^{i} t(o, l_q)_\mu \geq \Delta_R \quad \text{mit} \quad \Delta_R \in \mathbb{R}^+ \tag{5.7}$$

Der Schwellenwert Δ_R verhindert eine zu starke Fragmentierung der Anwendung, indem er die Replikation auf Objekte mit einer hohen Anfragelast beschränkt.

Eine weitere Ursache für die Fragmentierung der Anwendung sind ungenutzte Replikate. Die Lastverwaltung erzeugt Replikate, um Anfragelast auf mehrere Rechner

zu verteilen. Wenn die Anfragelast wieder sinkt, kann es vorkommen, daß Replikate nicht mehr genutzt und damit überflüssig werden. Um eine zu starke Fragmentierung der Anwendung durch ungenutzte Replikate zu vermeiden, wird ein weiterer Schwellenwert Δ_G eingeführt. Ein Replikat, dessen Last unter den Schwellenwert Δ_G fällt, wird entfernt, sofern es nicht das einzige Element seiner Replikatgruppe ist.

$$\frac{1}{i} \cdot \sum_{\mu=1}^{i} z(o, l_q)_\mu \leq \Delta_G \quad \text{mit} \quad \Delta_G \in \mathbb{R}^+ \tag{5.8}$$

Bereits in Kapitel 2.4 wurde erläutert, daß die Garbage Collection problematisch ist, da man wegen der Offenheit verteilter objektorientierter Systeme nicht feststellen kann, wieviele Referenzen auf ein Objekt existieren. In Bezug auf die Replikation stellt dies jedoch kein Problem dar, da mindestens ein Replikat bestehen bleibt und die Objektreferenz dadurch ihre Gültigkeit behält. Die Garbage Collection ist jedoch problematisch in Hinblick auf die strikte Einhaltung der statischen Bindung, da man nicht sicherstellen kann, daß ein Replikat tatsächlich nicht mehr genutzt wird. Die statische Bindung stellt jedoch nur eine Optimierung für die Replikation dar und hat keine Auswirkung auf die Konsistenz und die Korrektheit der Anwendung. Deshalb wird an dieser Stelle die statische Bindung dem Wunsch nach Garbage Collection untergeordnet. Bei einer sehr geringen Arbeitslast eines Replikats ist zu erwarten, daß der Nutzen der Garbage Collection größer ist, als die möglichen Einbußen durch die Mißachtung der statischen Bindung.

5.3.3 Strategie zur Lastzuweisung

Als Mechanismus zur Lastzuweisung verwendet dieses Lastverwaltungskonzept die Initialplazierung. Bei der Initialplazierung weist die Lastbewertung einem neu zu erzeugenden Objekt einen Rechner als Ausführungseinheit zu. Die Vorgehensweise ist dabei dieselbe, wie bei der Lastverschiebung, d.h. es muß der Zielrechner identifiziert werden, bei dem die Lastzuweisung die kleinste Bewertungszahl zur Folge hat. Im Gegensatz zur Lastverschiebung liegt zum Zeitpunkt der Initialplazierung keine Lastinformation über das zu erzeugende Objekt vor, so daß die Transformationsfunktion nicht verwendet werden kann. Deshalb muß für die Initialplazierung eine vereinfachte Bewertungszahl entwickelt werden. Zu diesem Zweck genügt es, lediglich die Lasten aller potentiellen Zielrechner zu betrachten und das Objekt auf dem Rechner mit der geringsten Last zu erzeugen. In Anlehnung an Gleichung 5.1 gehen dabei alle Ressourcen gleichgewichtet in die Bewertungszahl ein. Dies führt schließlich zur Definition der Bewertungszahl b:

$$b = \frac{1}{i} \cdot \sum_{\mu=1}^{i} r_{z\,\mu} \tag{5.9}$$

Bei der Initialplazierung wird ein neu zu erzeugendes Objekt folglich auf dem Rechner erzeugt, der die kleinste Bewertungszahl (b) und damit auch die geringste Last hat.

5.4 Realisierung der Kenngrößen des Lastmodells

In den vorangegangenen Abschnitten wurde die Lastverwaltungsstrategie anhand des zuvor definierten Lastmodells beschrieben. Wegen des hohen Abstraktionsgrades des Lastmodells war dies möglich, ohne zuvor die Realisierung seiner einzelnen Kenngrößen festzulegen. Dies gewährleistet eine flexible und allgemein anwendbare Lastverwaltungsstrategie. Im folgenden werden die einzelnen Kenngrößen des Lastmodells für ein konkretes verteiltes System und die darin verfügbaren Ressourcen festgelegt.

5.4.1 Rechnerlastvektoren

Die wichtigsten Ressourcen auf der Ebene des Rechensystems sind der Prozessor, das Netzwerk, der Arbeitsspeicher (Speicher) und der Hintergrundspeicher. Die entsprechenden Lasten werden als Prozessor-, Netzwerk-, Speicher- und Ein-/Ausgabelast bezeichnet.

Der Prozessor ist die zentrale Komponente eines jeden Rechners. Insofern kommt der Prozessorlast bei Lastverwaltung eine besondere Bedeutung zu. Beim Prozessor handelt es sich um eine exklusiv benutzbare Ressource, die das Betriebssystem den Verbrauchern im Time-Sharing-Verfahren zuteilt. Die maximale Kapazität ist auf einen rechnenden Prozeß beschränkt. Die nachgefragte Kapazität kann durch die Anzahl der Prozesse bzw. Threads, die auf die Zuteilung des Prozessors warten, ausgedrückt werden. In UNIX-Betriebssystemen beschreibt der sogenannte `avenrun`-Wert die Prozessorlast anhand der durchschnittlichen Länge der Prozessorwarteschlange [37, 63]. Der `avenrun`-Wert sagt aus, wieviele Prozesse bzw. Threads eines Rechners auf die Zuteilung des Prozessors warten. Dabei wird lediglich die Anzahl der Prozesse betrachtet, jedoch nicht ihr tatsächlicher Bedarf an Rechenzeit. Die Meßintervalle zur Erfassung und Berechnung der Prozessorlast sind im Vergleich zum Rechenzeitbedarf der Prozesse sehr klein, was diesen Nachteil nahezu aufwiegt. Der `avenrun`-Wert wird vom Betriebssystem über Zeitintervallen von einer, fünf und zehn Minuten geglättet, um starke Schwankungen durch vereinzelte Ausreißer zu verhindern. Wegen dieser Glättung reagiert dieser Wert mit einer entsprechenden Zeitverzögerungen auf Laständerungen. Wie eine Vielzahl von Arbeiten zeigten, ist der `avenrun`-Wert sehr gut geeignet, um die Prozessorlast abzubilden [30, 109, 73].

Eine weitere, wichtige Ressource in verteilten Systemen ist das Netzwerk, über das die Kommunikation abgewickelt wird. Der ausschlaggebende Faktor für die Auslastung des Netzwerks ist seine Übertragungskapazität. Die Netzwerklast ist das Verhältnis aus nachgefragter und maximaler Übertragungskapazität. In gängigen Betriebssystemen wird die Auslastung des Netzwerks jedoch als Verhältnis der zugeteilten und der maximal verfügbaren Übertragungskapazität dargestellt. Dies ist nach Definition 3.2 keine Last, da anstelle der Nachfrage nur die befriedigte Nachfrage betrachtet wird. Mit diesem Verfahren ist es nicht möglich Überlasten darzustellen, was jedoch eine Grundvoraussetzung für die Lastverwaltung ist. Der Grund für diese Vorgehensweise liegt darin, daß gängige Betriebssysteme die Auslastung des Netzwerks am Übertra-

gungsmedium messen. An dieser Stelle liegt nur Information über die bereits zugeteilte Übertragungskapazität vor. Ein geeignetes Verfahren müßte in das Netzwerkprotokoll integriert sein, da nur dort Information über die nachgefragte Übertragungskapazität vorliegt. Die Nachfrage aller Rechner, die das Verbindungsmedium gemeinsam nutzen, könnte dann summiert und in Relation zur maximalen Kapazität des Mediums gesetzt werden. Dieses Verfahren ist zwar aufwendiger, würde jedoch einen echten Lastwert liefern.

Das hier vorgestellte Lastmodell bietet keine Möglichkeit zur Darstellung der Netzwerktopologie. Dadurch ist die Sichtweise der Lastverwaltung auf die Betrachtung einzelner Rechner und ihrer Schnittstellen zum Verbindungsnetzwerk beschränkt. Globale Aspekte wie die Wegewahl (Routing) werden mit diesem Lastmodell nicht erfaßt, da dies bereits auf der Ebene der Netzwerkprotokolle geschieht [42, 52]. Die Lastverwaltung kann lediglich für eine gleichmäßige Auslastung unterschiedlicher Netzsegmente sorgen, indem sie Objekte aus einem überlasteten Segment in ein unterlastetes umverteilt. Die Steuerung der Datenströme zwischen den Netzwerksegmenten ist Aufgabe der Netzwerkprotokolle. Eine engere Verzahnung dieser Problemstellungen könnte für beide Seiten vorteilhaft sein und sollte gegebenenfalls in zukünftigen Arbeiten untersucht werden.

Eine gewisse Sonderstellung unter den Ressourcen eines verteilten Systems nimmt der Speicher ein. Der Speicher ist hierarchisch organisiert, wobei sich die einzelnen Hierarchiestufen sowohl in der Zugriffszeit als auch in der Menge des verfügbaren Speichers unterscheiden. Verallgemeinernd kann man sagen: Je geringer das Speichervolumen ist, desto kürzer ist auch die Zugriffszeit. Auf der obersten Ebene befindet sich der Prozessor-Cache, der über ein sehr geringes Speichervolumen und eine sehr kurze Zugriffszeit verfügt. Darunter folgt der Hauptspeicher mit mittlerem Speichervolumen und Zugriffszeit. Auf unterster Ebene findet man den Auslagerungsspeicher, der ein sehr großes Speichervolumen aber auch sehr lange Zugriffszeiten hat. Je nach Rechnerarchitektur können noch weitere Hierarchiestufen identifiziert werden. Für die Lastverwaltung sind insbesondere der Haupt- und der Auslagerungsspeicher von Bedeutung. Wenn die Nachfrage nach Hauptspeicher das Angebot überschreitet, muß auf den Auslagerungsspeicher zurückgegriffen werden, was schließlich langsamere Speicherzugriffe der Anwendung zur Folge hat. Die Speicherlast ergibt sich aus dem Verhältnis von nachgefragtem Speicher und vorhandenem Hauptspeicher:

$$\text{Speicherlast} = \frac{\text{nachgefragter Speicher}}{\text{vorhandener Hauptspeicher}} \tag{5.10}$$

Der nachgefragte Speicher beinhaltet sowohl den bereits verbrauchten Haupt- als auch den belegten Auslagerungsspeicher. Der Prozessor-Cache kann in diesem Zusammenhang vernachlässigt werden, da er vom Prozessor selbst verwaltet wird und somit nicht der Kontrolle der Lastverwaltung unterliegt. Der Vorteil dieser Betrachtungsweise liegt darin, daß trotz der hierarchischen Organisation des Speichers eine geschlossene Darstellung der Speicherlast möglich ist.

Eine weitere Ressource ist der Hintergrundspeicher, über den die Ein-/Ausgabe abgewickelt wird. In verteilten Systemen ist die Ein-/Ausgabe häufig über ein verteiltes Dateisystem oder eine Datenbank realisiert, um die Ortstransparenz beim Zugriff

auf den Hintergrundspeicher zu gewährleisten. Die Adressierung der entsprechenden Dienste erfolgt über das Netzwerk. Die Betriebssystem- oder Datenbankdienste, die den tatsächlichen Zugriff auf den Hintergrundspeicher abwickeln, sind nicht Bestandteil der verteilten Anwendung und unterliegen damit nicht der Kontrolle der Lastverwaltung. Aus diesem Grund wird der Hintergrundspeicher hier nicht betrachtet. In diesem Zusammenhang sei auf Literatur zur Lastverwaltung im Bereich der Datenbanksysteme verwiesen [108, 48].

Die Lastwerte der einzelnen Ressourcen werden nun zu einem Rechnerlastvektor zusammengefaßt:

$$\begin{aligned} r = (CPU, NET, MEM) \text{ mit } r \in \mathcal{R}, \\ r_{CPU} \quad & \text{als Lastwert des Prozessors,} \\ r_{NET} \quad & \text{als Lastwert des Netzwerks und} \\ r_{MEM} \quad & \text{als Lastwert des Speichers} \end{aligned} \tag{5.11}$$

Die Netzwerklast wurde zwar in den Rechnerlastvektor aufgenommen, kann aber bei der weiteren Realisierung des Lastmodells nicht betrachtet werden, da gängige Betriebssysteme keinen geeigneten Wert zur Verfügung stellen.

5.4.2 Leistungsvektoren

Leistungsvektoren sind nötig, um die Leistungsfähigkeit heterogener Rechner und ihrer Ressourcen vergleichen zu können. Ein Leistungsvektor beschreibt die maximale Kapazität der Ressourcen eines Rechners. Die maximalen Kapazitäten des Netzwerks und des Speichers können einfach anhand der entsprechenden Hardware-Parameter ermittelt werden. Beim Prozessor hingegen ist die Bestimmung der Leistungsfähigkeit wesentlich schwieriger. Die Leistung des Prozessors besteht im wesentlichen in der Ausführung von Programm-Code. Zur Messung der Leistungsfähigkeit werden sogenannte Benchmarks verwendet [145]. Ein Benchmark ist ein Programm, dessen Ausführungszeit als Maß für die Leistungsfähigkeit eines Rechners dient. Ein weit verbreiteter Benchmark zur Bewertung der Prozessorleistung ist der SPEC CPU2000 von der Standard Performance Evaluation Corporation (SPEC) [47]. Der SPEC CPU2000 besteht aus einer Sammlung realistischer Programme, deren Ausführungszeiten maßgeblich vom Prozessor (SPECint2000) bzw. der Gleitkommaeinheit (SPECfp2000) abhängen.

Aus den Leistungswerten der einzelnen Ressourcen kann nun der Leistungsvektor eines Rechners gebildet werden:

$$\begin{aligned} l = (CPU, NET, MEM) \text{ mit } l \in \mathcal{L}, \\ l_{CPU} \quad & \text{als Leistungswert des Prozessors,} \\ l_{NET} \quad & \text{als maximale Kapazität des Netzwerks in } \tfrac{\text{Byte}}{\text{Sekunde}} \text{ und} \\ l_{MEM} \quad & \text{als maximale Kapazität des Hauptspeichers in Byte} \end{aligned} \tag{5.12}$$

5.4.3 Objektlastvektoren

Objektlastvektoren sind die bestimmende Kenngröße der Anwendungsebene. Sie beschreiben die Arbeitslast von Objekten durch eine Menge von Einzelwerten, die in enger Beziehung zu den Lastwerten des Rechensystems stehen:

$$
\begin{aligned}
&o = (RR, RPT, NET, MEM) \text{ mit } o \in \mathcal{O}, \\
&\quad o_{RR} \quad \text{als Anfragerate des Objekts in } \tfrac{\text{Anfragen}}{\text{Sekunde}}, \\
&\quad o_{RPT} \quad \text{als Rechendauer einer Anfrage in Sekunden,} \\
&\quad o_{NET} \quad \text{als Kommunikationsvolumen einer Anfrage in Byte und} \\
&\quad o_{MEM} \quad \text{als Speicherbedarf des Objekts in Byte}
\end{aligned}
\tag{5.13}
$$

Diese Werte können, mit Ausnahme des Speicherbedarfs, auf der Ebene des verteilten Ablaufsystems ermittelt werden. Der Broker, als Vermittler zwischen Clients und Objekten, hat Kenntnis über die Anfragerate eines Objekts sowie über die Rechendauer und das Kommunikationsvolumen von Anfragen. Die Information über den Speicherbedarf stellt in der Regel das Betriebssystem bereit. Gängige Betriebssysteme arbeiten jedoch prozeßorientiert und liefern deshalb lediglich Information über Server-Prozesse aber nicht über einzelne Objekte. Aus diesem Grund kann der Speicherbedarf von Objekten bei der Realisierung des Lastmodells nicht berücksichtigt werden.

5.4.4 Die Transformationsfunktion

Die Transformationsfunktion stellt einen Bezug zwischen der Lastinformation der Anwendungsebene und der Lastinformation des Rechensystems her, indem sie Objekt- auf Rechnerlastvektoren abbildet:

$$
\begin{aligned}
&t : \mathcal{O} \times \mathcal{L} \rightarrow \mathcal{R} \\
&t(o, l) = \left(o_{RR} \cdot o_{RPT}, \frac{o_{NET}}{l_{NET}}, \frac{o_{MEM}}{l_{MEM}} \right)
\end{aligned}
\tag{5.14}
$$

Die Prozessorlast, die ein Objekt erzeugt, ergibt sich aus der Anzahl der gleichzeitig abzuarbeitenden Anfragen. Gängige Ablaufsysteme erreichen die nebenläufige Abarbeitung, indem sie jeder Anfrage einen Betriebssystem-Thread zur Abarbeitung zuweisen. Daraus ergibt sich dann die Prozessorlast eines Objekts. Die Anzahl der gleichzeitig abgearbeiteten Anfragen entspricht schließlich dem Produkt der Anfragerate und der Rechendauer einer Anfrage. Einfacher ist die Berechnung der Netzwerk- und der Speicherlast eines Objekts. Diese können direkt aus der Ressourcennachfrage des Objekts und der maximalen Kapazität der entsprechenden Ressource gewonnen werden.

5.5 Zusammenfassung

In diesem Kapitel wurde ein Lastmodell für verteilte objektorientierte Systeme vorgestellt, das die einzelnen Kenngrößen beschreibt, die ein Lastbewertungsalgorithmus zur Entscheidungsfindung heranziehen kann. Dabei handelt es sich um ein

mehrschichtiges Modell, das sowohl die Ebene des Rechensystems als auch die Anwendungsebene betrachtet. Eine Innovation dieses Lastmodells ist die Transformationsfunktion, die es erlaubt, einen Bezug zwischen der Lastinformation der Anwendungsebene und der Lastinformation des Rechensystems herzustellen, ohne dabei Kenntnis über die Realisierung des Lastmodells und seiner einzelnen Kenngrößen zu haben. Bisher verfügbare Lastmodelle haben keine explizite Transformationsfunktion, sondern ermitteln den Einfluß der Anwendung auf die Last des Rechensystems direkt anhand von Lastwerten [109, 71]. Dies führt dazu, daß die Lastbewertungsstrategie abhängig wird von der konkreten Ausprägung der betrachteten Lastwerte. Kernpunkt dieses Lastmodells ist jedoch die strikte Trennung der Lastbewertungsstrategie und ihrer Realisierung. Damit ist es möglich, Lastbewertungsstrategien zu entwickeln, ohne dabei konkrete Lastwerte betrachten zu müssen. Dies gewährleistet, daß die Strategien flexibel und allgemein anwendbar sind. Um die prinzipielle Vorgehensweise zur Entwicklung konkreter Lastbewertungsstrategien aufzuzeigen, wurde ein Vorgehensmodell für dieses Lastmodell entwickelt. Das Vorgehensmodell basiert auf fiktiven Lastverteilungsaktionen, die eine konkrete Lastbewertungskomponente anhand ihrer Auswirkungen auf die Lastsituation des Rechensystems bewerten kann. Mit Hilfe der fiktiven Lastverteilungsaktionen ist es möglich, Strategien zu entwickeln, welche die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen erlauben. Damit grenzt sich dieses Lastmodell deutlich von anderen Ansätzen ab, die sich, wie in Kapitel 4.4 beschrieben wurde, auf wenige Lastverteilungsmechanismen beschränken. Anhand dieses Vorgehensmodells wurde schließlich eine Strategie zur Lastbewertung in verteilten objektorientierten Systemen entwickelt. Die Strategie ist leicht nachvollziehbar, da sie direkt aus den Anforderungen der Lastbalancierung abgeleitet ist. Hier zeigt sich ein wesentlicher Vorteil eines formal definierten Lastmodells: Durch die mathematisch exakte Darstellung der Lastbewertungsstrategie ist es möglich Aussagen, wie z.B. die Bestimmung des Verteilungswertes bei der Replikation, formal aus der Strategie abzuleiten. Dies stellt einen wesentlichen Vorteil gegenüber Systemen dar, deren Lastbewertungsstrategie auf der Anwendung einzelner Heuristiken basiert. Das Kapitel schließt mit der Realisierung des Lastmodells und seiner einzelnen Kenngrößen.

6. Das Lastverwaltungssystem LMC

In den vorangegangenen Kapiteln wurde schrittweise ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen entwickelt. Dieses Kapitel beschreibt die Realisierung des Konzepts für die Common Object Request Broker Architecture (CORBA). Für CORBA als Grundlage einer Implementierung sprechen mehrere Gründe: CORBA hat sich mittlerweile zu einer Art Industriestandard für die Entwicklung verteilter objektorientierter Anwendungen entwickelt. Darüber hinaus bietet CORBA ein standardisiertes und gut dokumentiertes Programmiermodell, welches das Paradigma der verteilten objektorientierten Programmierung konsequent und durchgängig verwirklicht.

Ziel dieses Kapitels ist die Vorstellung des Lastverwaltungssystems LMC (Load Managed CORBA) und seiner grundlegenden Konzepte. Das Hauptaugenmerk bei der Entwicklung von LMC ist auf die nahtlose Integration der Lastverwaltung in das CORBA-Programmiermodell gerichtet. LMC basiert auf der CORBA-Implementierung JacORB [13], die an der Freien Universität Berlin entwickelt wird. JacORB unterstützt die Programmiersprache Java und ist im Quelltext erhältlich. Das Lastverwaltungssystem LMC soll aufzeigen, wie das hier entwickelte Lastverwaltungskonzept für ein konkretes Programmiermodell umgesetzt werden kann. Darüber hinaus bildet die Implementierung die Grundlage für die Evaluierung des Lastverwaltungskonzepts im nachfolgenden Kapitel.

6.1 Grundlagen

Zunächst werden einige weiterführende Eigenschaften des Programmiermodells von CORBA erläutert, die in Kapitel 2.4.1 nur kurz angesprochen wurden.

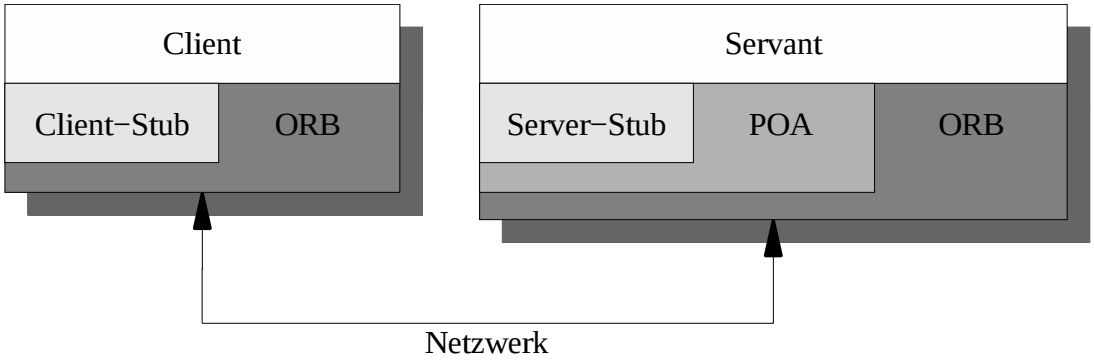
6.1.1 Aufbau einer CORBA-Anwendung

CORBA-Anwendungen sind nach dem Client-Server-Prinzip aufgebaut, das bereits in Kapitel 2.1.3 beschrieben wurde. Der Server ist ein Subsystem, das Objekten als

Laufzeitumgebung dient. In der Regel ist ein Server als Prozeß realisiert, der von einem Server-Rechner ausgeführt wird. Objekte wiederum verfügen über Methoden, die Clients durch entfernte Methodenaufrufe anfragen können.

Für den Entwickler stellt CORBA eine Reihe von Schnittstellen zur Verfügung, um den Client und den Server bzw. seine Objekte an das verteilte Ablaufsystem anzubinden. Abbildung 6.1 zeigt den Aufbau einer typischen CORBA-Anwendung und ihre einzelnen Komponenten. Client- und Server-Stubs werden von einem IDL-Compiler

Abbildung 6.1 Schematische Darstellung des Aufbaus einer CORBA-Anwendung



erzeugt und dienen als lokale Stellvertreter für die Methoden eines entfernten Objekts. Sie ermöglichen damit die transparente Ausführung entfernter Methodenaufrufe. Stubs kommen nur bei statischen Methodenaufrufen zum Einsatz; bei dynamischen Methodenaufrufen greift der Entwickler direkt auf die Schnittstellen der CORBA-Klassenbibliothek zu. Diese besteht hauptsächlich aus zwei Klassen, dem Object Request Broker (ORB) und dem Portable Object Adapter (POA). Die ORB-Klasse bildet den Kern der CORBA-Klassenbibliothek. Sie beinhaltet Funktionalität zur Verwaltung von Objektreferenzen und stellt die Infrastruktur für die Client-Server-Kommunikation zur Verfügung. Dazu zählt beispielsweise die Adressierung von entfernten Objekten, die Konvertierung der Parameter und die Datenübertragung bei Methodenaufrufen. Darüber hinaus verfügt der ORB über Funktionalität zum Instantiieren des Portable Object Adapters. Mit Hilfe des POAs kann der Entwickler die Abarbeitung von Methodenaufrufen auf der Server-Seite steuern. Die Parametrisierung des POAs erfolgt mittels spezieller Attribute, sogenannter Policies. Der POA ist verantwortlich für die Erzeugung von Objektreferenzen, die Steuerung des Objekt-Lebenszyklus und die Abarbeitung von Anfragen. Innerhalb eines Servers können mehrere Instanzen des POAs existieren. Dies ermöglicht dem Entwickler eine individuelle Steuerung der einzelnen Objekte. Damit kann man beispielsweise sowohl Objekte mit sequentieller als auch Objekte nebenläufiger Abarbeitung in einem Server-Prozeß realisieren. Wenn eine Anfrage den ORB des Servers erreicht, gibt sie dieser an den POA weiter, der wiederum die entsprechende Methode der lokalen Implementierung des Objekts aufruft. Die programmiersprachliche Implementierung eines Objekts wird in der CORBA-Terminologie als Servant bezeichnet. In der Praxis ist ein

Servant meist eine Instanz einer Klasse, die in einer bestimmten Programmiersprache geschrieben wurde. Ein CORBA-Objekt hingegen ist eine adressierbare Verwaltungseinheit, die durch unterschiedliche Servants realisiert sein kann. Die Trennung von Servant und Objekt verhindert, daß die programmiersprachliche Implementierung eines Objekts direkt adressiert wird, was für die Gewährleistung der Zugriffstransparenz unerläßlich ist.

6.1.2 Transparenz in CORBA

Eine der herausragenden Eigenschaften von verteilten objektorientierten Systemen ist die Transparenz und insbesondere die Orts- und die Zugriffstransparenz, die bereits in Kapitel 4.1 erläutert wurden. In Kapitel 4.2.4 wurde daraus für den Bereich der Lastverwaltung die Forderung nach Migrations- und Replikationstransparenz abgeleitet.

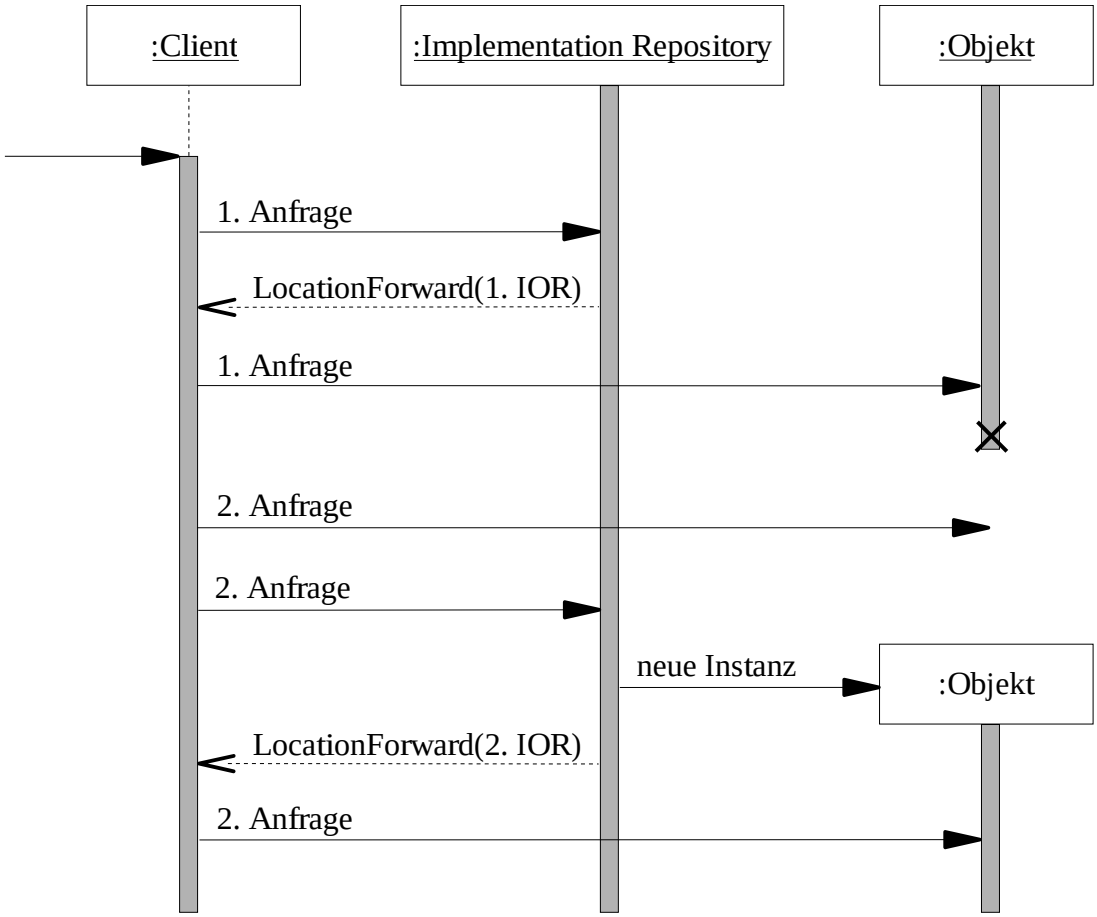
Die Zugriffstransparenz erreicht CORBA durch das General Inter-ORB-Protocol (GIOP). GIOP ist ein abstraktes Protokoll auf dem konkrete Protokolle wie beispielsweise das Internet Inter-ORB-Protocol (IIOP) aufsetzen. Diese Protokolle standardisieren die Kommunikation zwischen Client und Server und gewährleisten dadurch, daß alle Objekte auf dieselbe Art und Weise angefragt werden, unabhängig von der Beschaffenheit des Rechensystems, der verwendeten Programmiersprache und ihrer Implementierung.

Die Forderung nach Ortstransparenz erfüllt CORBA durch die Verwendung der Interoperable Object References (IOR). IORs dienen als Objektreferenzen, d.h. sie ermöglichen den Clients die Adressierung von entfernten Objekten. Der CORBA-Standard [98] legt fest, daß IORs ausschließlich vom ORB und nicht von den Clients selbst interpretiert werden. Dadurch ist sichergestellt, daß der Ausführungsort eines Objekts seinen Clients nicht bekannt ist.

Der CORBA-Standard bietet darüber hinaus auch Mechanismen zur Realisierung der Migrations- und der Replikationstransparenz [46]. Zu diesem Zweck wird je nach Lebensdauer zwischen persistenten und transienten Objektreferenzen unterschieden. Die Gültigkeit transienter Objektreferenzen ist auf die Lebensdauer des jeweiligen Objekts beschränkt. Persistente Objektreferenzen sind dauerhaft, d.h. sie können über die Lebensdauer eines Objekts hinaus verwendet werden. Dadurch kann man eine eingeschränkte Form der Garbage Collection verwirklichen. Objekte mit persistenter Referenz, die über einen längeren Zeitraum keine Anfragen erhalten haben, können entfernt und bei Bedarf wieder erzeugt werden. Dieser Vorgang ist für Clients transparent, da sie weiterhin über dieselben Referenzen auf die entsprechenden Objekte zugreifen können. Zur Umsetzung dieses Konzepts bietet CORBA den Mechanismus der Anfrageumleitung (Location Forwarding) an [98, 46]. Abbildung 6.2 zeigt das Ablaufprotokoll einer Anfrageumleitung. Persistente Objektreferenzen verweisen stets auf das Implementation Repository. Dies ist eine zentrale Instanz einer jeden CORBA-Domäne[1], mit welcher der ORB die Abbildung von persistenten Objektreferenzen auf Objekte bewerkstelligt. Die erste Anfrage eines

[1] Als Domäne bezeichnet man den Wirkungsbereich eines ORBs.

Abbildung 6.2 Das Ablaufprotokoll einer Anfrageumleitung



Clients ist nicht an das betreffende Objekt, sondern an das Implementation Repository gerichtet. Dieses beantwortet die Anfrage mit einer `LocationForward`-Exception, welche eine transiente Referenz auf das betreffende Objekt beinhaltet. Die Laufzeitumgebung des Clients nimmt diese Exception entgegen und verwendet im folgenden die darin enthaltene Objektreferenz. Anschließend sendet die Laufzeitumgebung des Clients die Anfrage erneut an das Objekt, das durch die transiente Referenz bezeichnet wird. Der Client verwendet die transiente Objektreferenz bis ein Fehler auftritt, der auf die Terminierung des betreffenden Objekts hindeutet. In diesem Fall verliert die transiente Referenz ihre Gültigkeit. Daraufhin benutzt der Client wieder die persistente Objektreferenz, die weiterhin gültig ist. Die nächste Anfrage ist folglich wieder an das Implementation Repository gerichtet. Das Implementation Repository erzeugt nun eine neue Instanz des betreffenden Objekts und gibt seine transiente Objektreferenz mittels einer `LocationForward`-Exception an den Client weiter. Der Mechanismus der Anfrageumleitung wird ausschließlich von der Laufzeitumgebung des Clients abgewickelt und ist dadurch transparent. Mit Hilfe der Anfrageumleitung kann die transparente Migration von Objekten realisiert werden, indem das Implementation

Repository für das neu zu erzeugende Objekt einen anderen Ausführungsort wählt. Zusammen mit der in CORBA gegebenen Ortstransparenz ist damit sichergestellt, daß jede Veränderung des Ausführungsortes eines Objekts seinen Clients verborgen bleibt. Nach Definition 4.1 ist dies exakt die Forderung der Migrationstransparenz. Die transparente Replikation von Objekten kann durch die Umleitung von Anfragen auf unterschiedliche Replikate bewerkstelligt werden. In diesem Fall erlangen die Clients eines Objekts keine Kenntnis darüber, daß mehrere Replikate existieren. Dies ist nach Definition 4.2 die Forderung der Replikationstransparenz.

6.1.3 Objektpersistenz

Die Objektpersistenz ermöglicht es bei der Migration und der Replikation von Objekten den Zustand des betreffenden Objekts zu erhalten, indem er vom Quell- auf das Zielobjekt übertragen wird. Bei zustandsbehafteten Objekten ist dies eine unabdingbare Voraussetzung für Gewährleistung der Äquivalenz von Quell- und Zielobjekt. Bei Objekten mit redundantem Zustand stellt die Zustandsübertragung, wie bereits in Kapitel 4.3.5 erläutert, eine Optimierung dar, mit welcher die Kosten für die Rekonstruktion des Zustandes eingespart werden können. Neben dem Begriff des Zustandes wird in Definition 4.7 auch der rekursive Zustand eines Objekts beschrieben. Der rekursive Zustand schließt neben dem Zustand eines Objekts auch den Zustand aller direkt oder indirekt referenzierten Objekte ein. Bei der Betrachtung des rekursiven Zustandes spricht man von Persistenz durch Erreichbarkeit.

In der Praxis gibt es eine Vielzahl von Persistenzmechanismen. Die einzelnen Verfahren unterscheiden sich hauptsächlich in ihrer Realisierung. Im folgenden werden einige ausgewählte Persistenzmechanismen vorgestellt und anhand ihrer Realisierungsebene kategorisiert:

- Betriebssystemebene: Persistente Betriebssysteme wie Multics [7], Clouds [21], Monads [56] und Grasshopper [22] bieten Persistenz auf der Ebene der Speicherverwaltung. Die Trennung von Arbeitsspeicher und Hintergrundspeicher (Dateisystem), wie sie in herkömmlichen Betriebssystemen üblich ist [137], wird in persistenten Betriebssystemen aufgehoben. Statt dessen kommt eine persistente virtuelle Speicherverwaltung zum Einsatz [140]. Diese gewährleistet die Dauerhaftigkeit des virtuellen Speichers, indem sie die einzelnen Seiten des Arbeitsspeichers in regelmäßigen Zeitabständen auf dem Hintergrundspeicher sichert. Dieser Vorgang ist für Anwendungen transparent. Persistente Betriebssysteme kommen hauptsächlich im Bereich der Fehlertoleranz zum Einsatz. Objektpersistenz im engeren Sinne kann damit nicht erreicht werden, da eine virtuelle Speicherverwaltung seitenorientiert und nicht objektorientiert arbeitet.

- Sprachebene: Persistenz auf Sprachebene bieten beispielsweise die Programmiersprachen Java und Python. Beide verwenden dazu den Mechanismus der Objektserialisierung [135, 104]. Damit kann der Zustand bzw. der rekursive Zustand eines Objekts dauerhaft gespeichert werden, um zu einem späteren Zeit-

punkt wieder ein Objekt daraus zu erzeugen. Die Objektserialisierung erfolgt in der Regel ohne das Zutun des Entwicklers und ist folglich transparent. Einen anderen Ansatz bei der Realisierung der Objektpersistenz verfolgt Texas [122]. Dabei handelt es sich um eine Klassenbibliothek, welche die Programmiersprache C++ um das Konzept der persistenten Objekte erweitert. Für die Erzeugung von persistenten Objekten wird analog zum sonst üblichen `new`-Operator der Operator `pnew` eingeführt. Die Speicherverwaltung von Texas gewährleistet die Dauerhaftigkeit der mit `pnew` erzeugten Objekte. Abgesehen von der Umbenennung des `new`-Operators ist auch diese Form der Objektpersistenz transparent.

- Dienstebene: Auf dieser Realisierungsebene ist die Funktionalität zur Gewährleistung der Objektpersistenz in separate Dienste ausgelagert, die von der eigentlichen Anwendung entkoppelt sind. CORBA stellt diesbezüglich zwei Dienste zur Verfügung: Den Persistent State Service[2] [93] und den Externalization Service [95]. Der Persistent State Service führt sogenannte Storage Objects ein, welche die persistenten Daten eines Objekts beinhalten. Die Beschaffenheit der Storage Objects kann mittels einer speziellen Modellierungssprache, der sogenannten Persistent State Definition Language, beschrieben werden. Der Persistent State Service sorgt dann für die Dauerhaftigkeit der Storage Objects. Eine andere Vorgehensweise findet man beim Externalization Service. In diesem Fall müssen persistente Objekte die Schnittstelle `Streamable` implementieren, mit deren Hilfe sie dann serialisiert werden können. Die Implementierung der Objektserialisierung bleibt beim Externalization Service dem Entwickler überlassen. Beide CORBA-Dienste sind hinsichtlich ihrer Transparenz stark eingeschränkt, da ein großer Teil der Arbeit auf den Entwickler abgewälzt wird. Eine weitere Möglichkeit zur Realisierung der Objektpersistenz auf Dienstebene stellen objektorientierte Datenbanken [69] dar. Diese bieten die Möglichkeit, den Zustand bzw. den rekursiven Zustand von Objekten in einer Datenbank zu speichern, um zu einem beliebigen späteren Zeitpunkt wieder darauf zugreifen zu können. Die Adressierung von persistenten Objekten erfolgt über spezielle Bezeichner, die von der Datenbank vergeben werden. Analog zu relationalen Datenbanken gewährleisten objektorientierte Datenbanken die Atomarität, die Konsistenz, die Isolierung und die Dauerhaftigkeit aller Transaktionen mit persistenten Objekten. Vertreter dieser Kategorie sind beispielsweise ObjectStore [19], Objectivity [86] und Versant [141, 142]. Die Verwendung objektorientierter Datenbanken ist für den Entwickler weitgehend transparent.

- Anwendungsebene: Eine weitere Möglichkeit zur Realisierung der Objektpersistenz bietet die Anwendung selbst. Auf der Anwendungsebene kann der Entwickler für die Persistenz von Objekten sorgen, indem er diese mit Funktionali-

[2] Der Persistent State Service wird in zukünftigen Versionen des CORBA-Standards den bisher gültigen Persistent Object Service [97] ersetzen.

tät zur Extraktion und Rekonstruktion ihres Zustandes ausstattet. Dies ist in der Regel mit erheblichem Mehraufwand verbunden und somit nicht transparent.

Dieser Überblick soll einen Eindruck von der Vielzahl der unterschiedlichen Persistenzmechanismen vermitteln, die für eine Implementierung der Lastverschiebung bzw. der Zustandsübertragung zur Verfügung stehen.

6.2 Anforderungen an die Implementierung

Neben den in Kapitel 4 beschriebenen konzeptionellen Aspekten muß die Implementierung der Lastverwaltung eine Reihe von allgemeinen Anforderungen erfüllen, die maßgeblichen Einfluß auf das Software-Design und die Realisierung der Lastverwaltungsfunktionalität haben. Im folgenden sind die wichtigsten Anforderungen aufgeführt:

- Nahtlose Integration in das CORBA-Programmiermodell: Die Schnittstellen der Lastverwaltung sollten so in das CORBA-Programmiermodell integriert sein, daß der Übergang für den Entwickler möglichst fließend verläuft. Dies kann durch eine Erweiterung der typischen CORBA-Schnittstellen, insbesondere des ORB und des POA, erreicht werden. Dies stellt sicher, daß die Schnittstellen der Lastverwaltung konform zum CORBA-Objektmodell sind.

- Geringer Mehraufwand für den Entwickler: Der Mehraufwand, der durch die Nutzung der Lastverwaltungsfunktionalität für den Entwickler entsteht, sollte so gering als möglich sein. Die Aufgabe des Entwicklers sollte sich auf die Parametrisierung der Lastverwaltung beschränken. Die Lasterfassung, die Lastbewertung die Lastverteilung muß die Lastverwaltung selbständig erledigen. Je geringer der Mehraufwand für den Entwickler ist, desto größer ist schließlich die Akzeptanz der Lastverwaltung.

- Gewährleistung der Migrations- und Replikationstransparenz: Wie bereits in Abschnitt 6.1.2 erläutert wurde, sind die Orts- und die Zugriffstransparenz wesentliche Bestandteile des CORBA-Objektmodells. Bei einer Erweiterung des Objektmodells um die Fähigkeit zur Migration und Replikation müssen darüber hinaus die Migrations- und die Replikationstransparenz gewährleistet sein. Einige grundlegende Mechanismen dazu stellt das CORBA-Programmiermodell bereits zur Verfügung. Diese müssen gegebenenfalls erweitert werden, um die statische Bindung, wie sie in Kapitel 4.3.6 gefordert ist, realisieren zu können.

- Unterstützung unterschiedlicher Persistenzmechanismen: Wie der vorangegangene Anschnitt zeigt, gibt es eine Vielzahl von Möglichkeiten zur Realisierung der Objektpersistenz. Die einzelnen Verfahren unterscheiden sich dabei sehr stark in ihrer Funktionalität und in dem Mehraufwand, den sie für den Entwickler verursachen. Die Lastverwaltung muß geeignete Schnittstellen bereitstellen, um dem Entwickler die Auswahl eines Verfahrens und gegebenenfalls seine Realisierung zu ermöglichen.

- Verwendung anerkannter Standards und Werkzeuge: Zur Realisierung der Lastverwaltungsfunktionalität sollten, soweit dies möglich ist, allgemein anerkannte Standards und Werkzeuge genutzt werden. Dies betrifft insbesondere Standardaufgaben wie zum Beispiel die Lasterfassung. Die Nutzung vorhandener Standards und Werkzeuge trägt einerseits zur Modularisierung bei und erleichtert andererseits die Implementierung der Lastverwaltung.

- Geringe Kosten der Lastverwaltung: Der Ressourcenverbrauch der Lastverwaltung sollte so gering als möglich sein, um den erwarteten Gewinn nicht zunichte zu machen. Dies bezieht sich hauptsächlich auf die Lastverteilung, da diese wahrscheinlich einen großen Teil der Kosten ausmacht. CORBA bietet bereits Mechanismen wie die Anfrageumleitung an, die eine kostengünstige Lastverteilung ermöglichen. Diese Verfahren sollten genutzt und gegebenenfalls erweitert werden.

- Skalierbarkeit der Lastverwaltung: Die Lastverwaltung sollte in verteilten Systemen unterschiedlicher Ausdehnung und Größe einsetzbar sein. In der Praxis sind CORBA-Systeme in sogenannte ORB-Domänen unterteilt, wobei eine ORB-Domäne den Wirkungsbereich eines ORBs bezeichnet. Die Aufteilung in einzelne ORB-Domänen ergibt sich in der Regel aus administrativen Beschränkungen. Da auch die Lastverwaltung diesen Beschränkungen unterliegt sollte der Wirkungsbereich der Lastbewertung deckungsgleich mit einer ORB-Domäne sein.

6.3 Realisierung der Lastverwaltungsfunktionalität

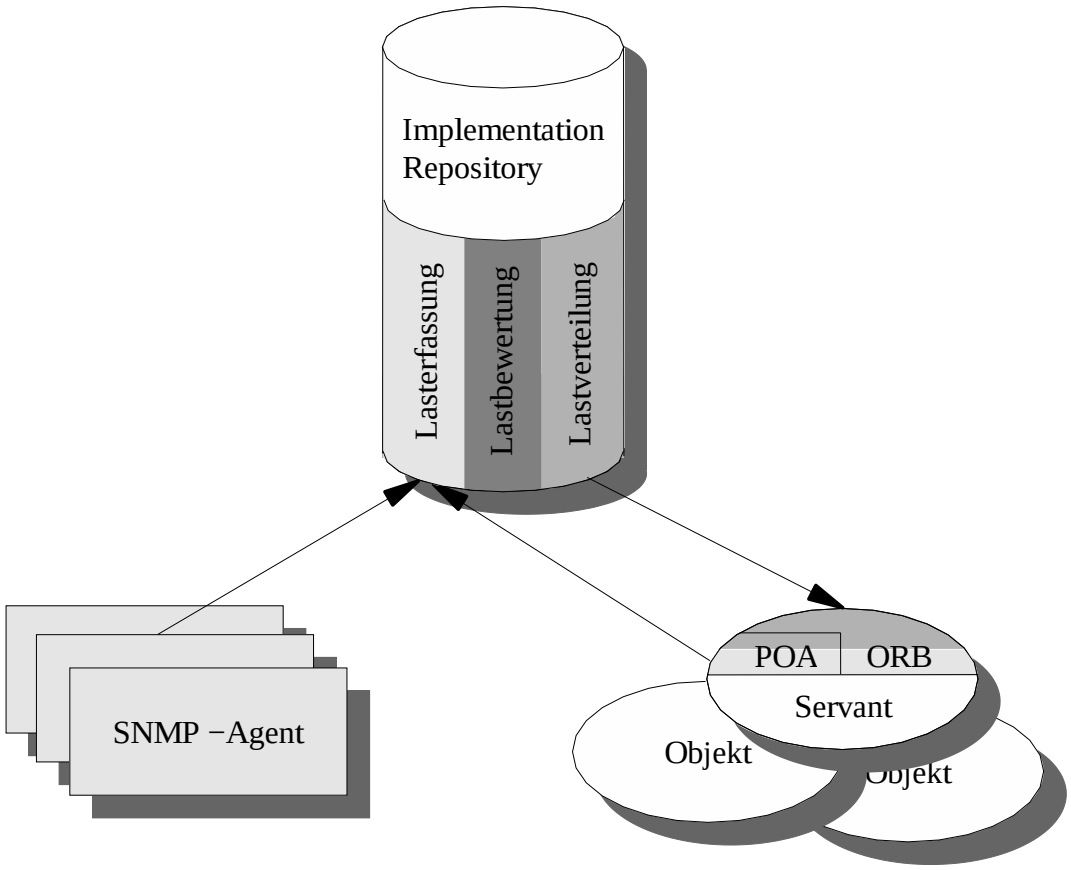
Die folgenden Abschnitte erläutern die Komponenten des Lastverwaltungssystems LMC, die Realisierung der grundlegenden Lastverwaltungsfunktionalität und die Integration in den CORBA-Standard.

6.3.1 Die Komponenten des Lastverwaltungssystem

Die Lastverwaltung gliedert sich, wie in Kapitel 3.1.2 erläutert wurde, in die Komponenten Lasterfassung, Lastbewertung und Lastverteilung. Abbildung 6.3 zeigt die einzelnen Komponenten des Lastverwaltungssystems LMC und ihre Interaktion mit dem zugrundeliegenden verteilten System.

Die zentrale Komponente von LMC ist die Lastbewertung, welche die in Kapitel 5.3 beschriebene Lastbewertungsstrategie umsetzt. Die Lastbewertung ist in das Implementation Repository integriert. Wie später zu sehen sein wird, ermöglicht dies eine enge örtliche und funktionale Koppelung der Lastbewertung und der übrigen Komponenten. Dies ist notwendig, um die Kosten für die Interaktion der einzeln Lastverwaltungskomponenten so gering als möglich zu halten. Der Wirkungsbereich der Lastbewertung ist deckungsgleich mit dem des Implementation Repository und erstreckt sich somit über eine ORB-Domäne. Aus der Sicht einer ORB-Domäne handelt

Abbildung 6.3 Das Lastverwaltungssystem LMC



es sich folglich um eine zentrale Komponente mit globaler Systemkenntnis und maximalem Wirkungsbereich. Betrachtet man jedoch ein verteiltes System mit mehreren ORB-Domänen, so verfügt LMC über eine gruppierte Lastbewertung mit partieller Systemkenntnis und begrenztem Wirkungsbereich. Für die Entscheidung, den Wirkungsbereich der Lastbewertung an die jeweilige ORB-Domäne zu koppeln sprechen sowohl konzeptionelle als auch technische Gründe. Einerseits gewährleistet dies die Skalierbarkeit der Lastverwaltung, da sehr große verteilte Systeme in mehrere ORB-Domänen unterteilt werden können. Andererseits sind zur Durchführung der Lasterfassung und der Lastverteilung administrative Privilegien nötig, die in der Regel nur innerhalb der jeweiligen ORB-Domäne zur Verfügung stehen. Insofern wäre ein Wirkungsbereich, der über eine ORB-Domäne hinausreicht nur eingeschränkt nutzbar. Unterschiedliche ORB-Domänen sind in der Praxis disjunkt, so daß hier auf eine Kooperation der einzelnen Lastbewertungskomponenten verzichtet wird. Die Lastbewertung erfolgt zeitgesteuert, d.h. der Regelkreis der Lastverwaltung wird in regelmäßigen Zeitintervallen durchlaufen.

Die Lasterfassung besteht aus drei Teilkomponenten. Die Hauptkomponente ist im Implementation Repository angesiedelt und sammelt in regelmäßigen Zeitabständen

Lastinformation über das Rechensystem und die Anwendungen. Die Lastinformation selbst wird von den anderen beiden Teilkomponenten bereitgestellt. Die Teilkomponente, welche die Lastinformation über die Anwendungen und ihre einzelnen Objekte liefert ist in den ORB und den POA integriert. Die Lastinformation der Anwendungsebene ist analog zu den Objektlastvektoren strukturiert, die in Kapitel 5.4.3 beschrieben wurden. Die dritte Teilkomponente ist für die Lastinformation des Rechensystems, d.h. die Leistungs- und Rechnerlastvektoren, verantwortlich. Zu diesem Zweck wird das Simple Network Management Protocol (SNMP) [75, 126] eingesetzt. SNMP ist ein weit verbreiteter Standard zur Verwaltung von Rechnernetzen, der den systemunabhängigen Zugriff auf Rechner- und Netzwerk-Ressourcen regelt. Alle Rechner und alle relevanten Netzwerkkomponenten verfügen über einen sogenannten SNMP-Agenten. Mit Hilfe der SNMP-Agenten kann sowohl steuernd als auch beobachtend auf die entsprechenden Ressourcen zugegriffen werden. Da es sich bei SNMP um einen erweiterbaren und flexiblen Standard handelt, können verfügbare Agenten leicht angepaßt werden, so daß sie Lastinformation entsprechend der Leistungs- und Rechnerlastvektoren bereitstellen. Alle Komponenten der Lasterfassung arbeiten auf der Ebene des Ablaufsystems. Dadurch entsteht kein Mehraufwand für den Entwickler, d.h. die Lasterfassung erfolgt transparent.

Die letzte Komponente des Lastverwaltungssystems ist die Lastverteilung. Sie ist in zwei Teileinheiten aufgespalten. Der eine Teil befindet sich im Implementation Repository und ist für die Initialplazierung und die Anfrageumleitung bei der Migration und der Replikation verantwortlich. Der zweite Teil ist in den ORB und den POA integriert und kümmert sich um das Erzeugen und Entfernen von Replikaten. Die Lastverteilung ist, wie schon die Lasterfassung, auf der Ebene des Ablaufsystems realisiert und somit transparent.

6.3.2 Objekt-Lebenszyklus

Der Lebenszyklus eines Objekts reicht von seiner Erzeugung, seiner Überführung in einen persistenten Zustand, seiner Rekonstruktion aus seinem persistenten Zustand, bis hin zu seiner Zerstörung. Einige dieser Aspekte sollten ursprünglich durch den Life Cycle Service [96] von CORBA abgedeckt werden. Der Life Cycle Service stellt eher ein Entwurfsmuster als einen Dienst im engeren Sinne dar, d.h. er bildet lediglich ein Gerüst anhand dessen der Entwickler die entsprechende Funktionalität implementieren kann. In der Praxis hat sich herausgestellt, daß die Implementierung konkreter Dienste anhand des Life Cycle Service sehr aufwendig und schwierig ist. Insbesondere die Realisierung von weiterreichender Funktionalität, wie beispielsweise der Objektpersistenz, erfordert eine engere Anbindung an das CORBA-Ablaufsystem als es mit einem externen Dienst möglich ist. Aus diesen Gründen findet der Life Cycle Service in der Praxis kaum Anwendung.

Wegen der Unzulänglichkeit des Life Cycle Service erweitert das Lastverwaltungssystem LMC den CORBA-Standard um Funktionalität zur Verwaltung des Objekt-Lebenszyklus. Dazu wird zuerst dem Implementation Repository die Schnittstelle `GenericFactory` hinzugefügt. Abbildung 6.4 zeigt die Schnittstellendefiniti-

Abbildung 6.4 Die Schnittstelle des Implementation Repository

```
module ImplementationRepository {
  interface GenericFactory {
    exception NoFactory {};
    exception NotRemovable {};

    boolean supports_key(in CORBA::RepositoryKey key);

    Object create_object(in CORBA::RepositoryKey key)
      raises(NoFactory);

    void remove_object(in Object obj)
      raises(NotRemovable);
  };
};
```

on in CORBA-IDL. In Anlehnung an das Factory Entwurfsmuster [34] können mit Hilfe der `GenericFactory` Objekte erzeugt und gegebenenfalls wieder entfernt werden. Die Methoden `create_object` und `remove_object` bilden dabei das Gegenstück zu den `new`- und `delete`-Operationen in objektorientierten Programmiersprachen. Objekte unterschiedlichen Typs werden mittels des sogenannten `RepositoryKey` identifiziert. Das Implementation Repository nutzt diesen Schlüssel, um einen Server-Prozeß ausfindig zu machen, der in der Lage ist Objekte vom angefragten Typ zu erzeugen. Ist kein geeigneter Server-Prozeß verfügbar, so erzeugt ihn das Implementation Repository automatisch.

Die Verwaltung des Objekt-Lebenszyklus erfordert auch eine Erweiterung des Portable Object Adapters (POA). Diese ist in Abbildung 6.5 dargestellt. Der POA erhält eine neue Policy, die `ControlFlowPolicy`, mit deren Hilfe der Entwickler angeben kann, wer den Objekt-Lebenszyklus kontrolliert. Der Wert `USER` besagt, daß der Entwickler den Lebenszyklus seiner Objekte kontrolliert, wie es bisher in CORBA üblich war. Durch Angabe des Wertes `SYSTEM` übergibt der Entwickler die Kontrolle über den Objekt-Lebenszyklus an das Ablaufsystem von CORBA. Dies ist eine Vorbedingung für den Einsatz der Lastverwaltung, da diese in der Lage sein muß, selbständig Objekte zu erzeugen und gegebenenfalls auch wieder zu zerstören. Die entsprechende Funktionalität muß der Entwickler bereitstellen, indem er eine `ServantFactory` implementiert. Die Schnittstelle der `ServantFactory` ist ähnlich gestaltet wie die der `GenericFactory`, mit dem Unterschied, daß die `ServantFactory` Servants und die `GenericFactory` Objekte betrachtet. Der POA kann Anfragen an ein Objekt automatisch an den entsprechenden Servant weiterleiten. Dazu wird die `RequestProcessingPolicy` um den Wert `USE_SERVANT_FACTORY` erweitert. Dieser Wert gibt an, daß der POA Anfragen, die sich auf ein bestimmtes Objekt beziehen, an den Servant weiterleitet, der bei der Erzeugung dieses Objekts verwendet wurde.

Abbildung 6.5 Erweiterung des POAs um die `ServantFactory`

```
module PortableServer
{
  ...

  enum RequestProcessingPolicyValue
  {
    ...
    USE_SERVANT_FACTORY
  };

  ...

  const CORBA::PolicyType CONTROL_FLOW_POLICY_ID = 23;

  enum ControlFlowPolicyValue
  {
    USER,
    SYSTEM
  };

  interface ControlFlowPolicy : CORBA::Policy
  {
    readonly attribute ControlFlowPolicyValue value;
  };

  interface ServantFactory
  {
    exception NoFactory {};

    typedef Object Cookie;

    CORBA::RepositoryKeySeq get_keys();

    Servant create_servant(in CORBA::RepositoryKey key,
                           out Cookie cookie)
      raises(NoFactory);

    void destroy_servant(in Servant servant,
                         in Cookie cookie);
  };

  ...
};
```

Durch das Zusammenwirken der `GenericFactory` und der `ServantFactory` können nun Objekte entsprechend dem Factory Entwurfsmuster erzeugt und wieder zerstört werden. Für die Lastverteilung muß darüber hinaus auch noch die Objektpersistenz betrachtet werden. Wie bereits in Abschnitt 6.1.3 beschrieben wurde, ist dazu ein Mechanismus nötig, mit dessen Hilfe man den Zustand eines Objekts extrahieren kann, um zu einem späteren Zeitpunkt wieder ein Objekt daraus zu erzeugen. Zu diesem Zweck wird die Schnittstelle des POAs um eine neue Policy, die sogenannte `PersistencePolicy`, erweitert. Abbildung 6.6 zeigt die entsprechende Schnittstelle. Die `PersistencePolicy` kann zwei Werte annehmen. `NO_PERSISTENCE` besagt, daß keine Mechanismen zur Gewährleistung der Objektpersistenz verfügbar sind. Durch Angabe des Wertes `USE_PERSISTENT_SERVANT_FACTORY` kann der Entwickler dem CORBA-Ablaufsystem mit Hilfe einer sogenannten `PersistentServantFactory` Zugriff auf den Objektzustand ermöglichen. Die `PersistentServantFactory` ist von der `ServantFactory` abgeleitet und erweitert diese um die Methoden `get_state` und `recreate_servant`. Mit Hilfe von `get_state` kann der Zustand eines Objekts bzw. seines Servants extrahiert werden, um daraus zu einem späteren Zeitpunkt mittels `recreate_servant` wieder einen Servant bzw. ein Objekt zu erzeugen. Die Realisierung dieser Methoden und damit auch die Auswahl eines geeigneten Persistenzmechanismus bleibt dem Entwickler überlassen.

6.3.3 Lastverteilung

Als Mechanismen der Lastverteilung nutzt das Lastverwaltungssystem LMC die Initialplazierung, die Migration und die Replikation von Objekten.

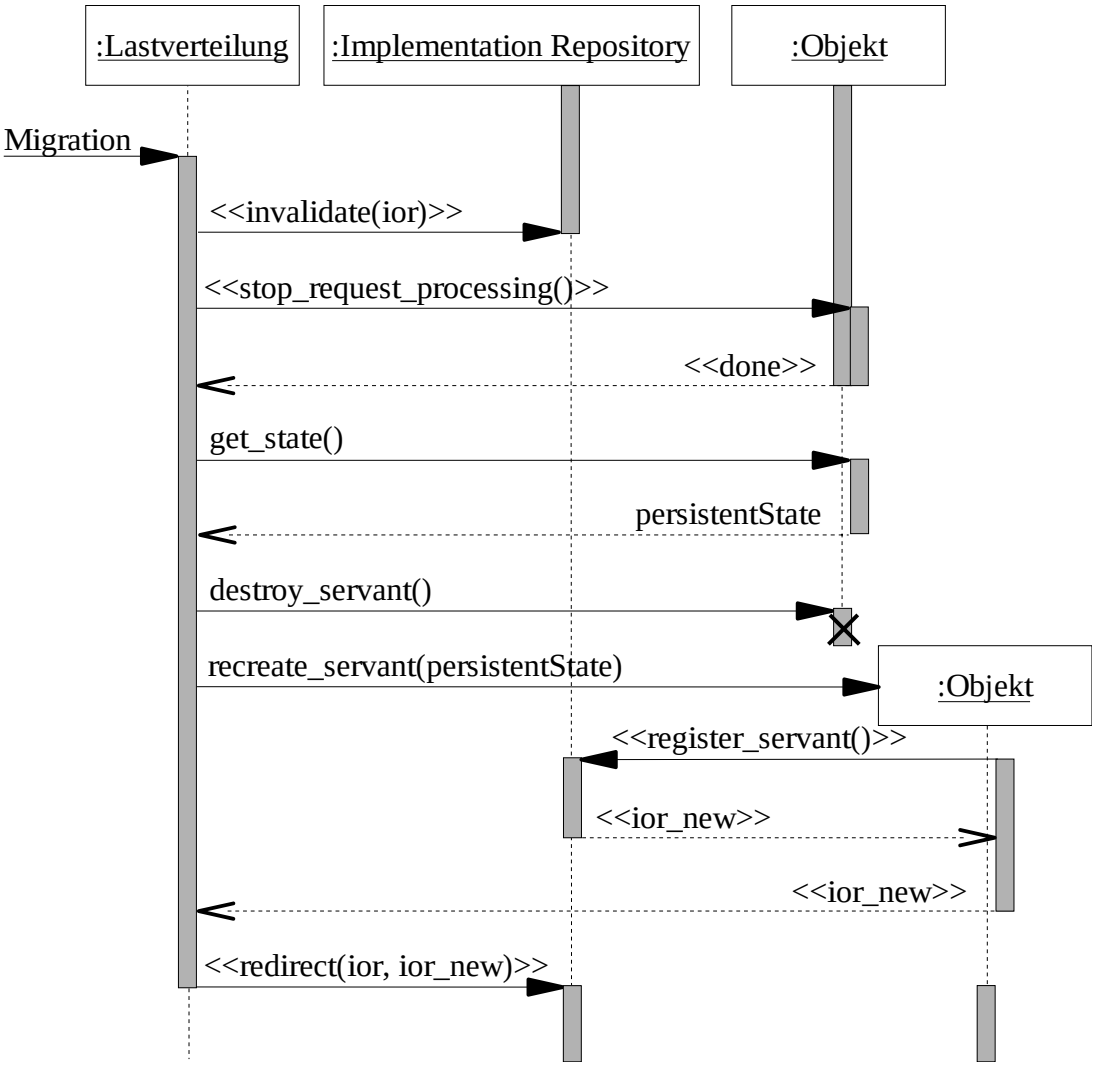
Die Realisierung der Initialplazierung setzt auf der `GenericFactory` auf. Bei jedem Aufruf der Methode `create_object` sucht die Lastbewertungskomponente nach einem geeigneten Ausführungsort für das zu erzeugende Objekt. Wegen der in CORBA von vornherein gegebenen Ortstransparenz ist auch die Initialplazierung für den Entwickler transparent.

Die Migration und Replikation von Objekten verläuft analog zu den in Kapitel 4.3.4 erläuterten Protokollen. Zur Realisierung dieser Protokolle benutzt die Lastverteilungskomponente die zuvor beschriebenen Erweiterungen des CORBA-Standards. Exemplarisch wird hier die Implementierung der Objektmigration herausgegriffen und näher erläutert. Abbildung 6.7 zeigt den Ablauf einer Migration als UML-Sequenzdiagramm. Um die Erweiterungen des CORBA-Standards und die internen Methoden der Lastverwaltung im Diagramm unterscheidbar zu machen, sind interne Methoden mit spitzen Klammern umgeben. Zuerst versetzt die Lastverteilung das zu migrierende Objekt in einen sicheren Zustand. Zu diesem Zweck wird das Implementation Repository veranlaßt jede weitere Anfrageumleitung zum betreffenden Objekt bis auf Widerruf zu verzögern. Anschließend teilt die Lastverteilung dem Objekt bzw. seinem POA mit, daß die Abarbeitung von Anfragen einzustellen ist, sobald alle aktiven Anfragen bearbeitet sind. Nachdem der sichere Zustand erreicht ist, extrahiert die Lastverteilung mittels der Methode `get_state` den Zustand des Objekts.

Abbildung 6.6 Erweiterung des POAs um die `PersistentServantFactory`

```
module PortableServer
{
  ...

  const CORBA::PolicyType PERSISTENCE_POLICY_ID = 24;

  enum PersistencePolicyValue
  {
    NO_PERSISTENCE,
    USE_PERSISTENT_SERVANT_FACTORY
  };

  interface PersistencePolicy : CORBA::Policy
  {
    readonly attribute PersistencePolicyValue value;
  };

  typedef sequence <any> ValueSeq;

  struct PersistentState
  {
    CORBA::RepositoryKey key;
    ValueSeq             value;
  };

  interface PersistentServantFactory : ServantFactory
  {
    exception InvalidState {};

    Servant recreate_servant(in PersistentState state,
                             out Cookie cookie)
      raises(NoFactory, InvalidState);

    PersistentState get_state(in Servant servant,
                              in Cookie cookie)
      raises(InvalidState);
  };

  ...
};
```

Abbildung 6.7 Realisierung der Objektmigration



Der Servant des Quellobjekts ist nun überflüssig und kann durch einen Aufruf von `destroy_servant` entfernt werden. Anschließend erzeugt die Lastverteilung mit Hilfe der Methode `recreate_servant` eine neue Instanz des Servants an einem Ausführungsort, der von der Lastbewertung vorgeschlagen wird. Der neue Servant registriert sich automatisch beim Implementation Repository und erhält dabei eine neue transiente Objektreferenz, die er anschließend der Lastverteilung übermittelt. Zum Schluß installiert die Lastverteilung beim Implementation Repository ein neues Ziel für die Anfrageumleitung an das betreffende Objekt. Alle weiteren Anfragen werden nun automatisch umgeleitet. Die Transparenz der Anfrageumleitung gewährleistet dabei auch die Migrationstransparenz.

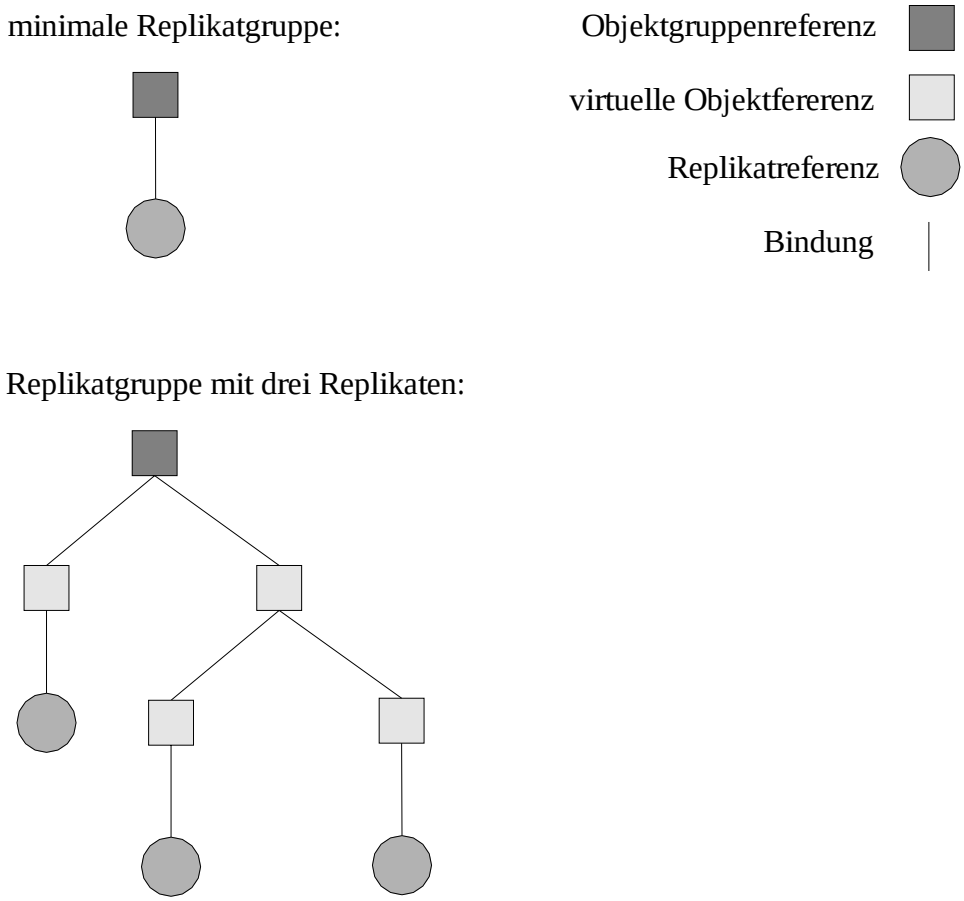
6.3.4 Statische Bindung

Im ursprünglichen Objektmodell von CORBA war die Replikation von Objekten nicht vorgesehen. Durch die Verabschiedung der Fault Tolerant CORBA Specification [92] wurde das Objektmodell schließlich um die Möglichkeit zur Erzeugung und Verwaltung von Replikatgruppen erweitert. Die Umsetzung dieses Konzepts orientiert sich jedoch stark an den Zielsetzungen und den Mechanismen der Fehlertoleranz und ist deshalb im Bereich der Lastverwaltung nur sehr eingeschränkt nutzbar. Bei der Lastverwaltung ist insbesondere die Bindung von Clients an Replikate von Bedeutung, die bereits in Kapitel 4.3.6 erläutert wurde. Bei der statischen Bindung muß gewährleistet sein, daß aufeinanderfolgende Anfragen eines Clients stets an dasselbe Replikat gerichtet sind. Darüber hinaus muß bei der Lastverschiebung sichergestellt werden, daß die Clients des Quellreplikats ausschließlich an die entsprechenden Zielreplikate gebunden werden. Mit dem ursprünglichen Mechanismus der Anfrageumleitung kann dies nicht erreicht werden, da Clients keine Identifikatoren besitzen. Damit kann das Ablaufsystem die Zuordnung zwischen Client und Replikat nicht dauerhaft speichern. Dies wäre jedoch nötig, um sicherstellen zu können, daß die Clients eines Quellreplikats bei der Lastverschiebung ausschließlich an die entsprechenden Zielreplikate gebunden werden. Die Realisierung der statischen Bindung erfordert folglich ein neues und erweitertes Konzept zur Verwaltung von Replikatgruppen.

Das Lastverwaltungssystem LMC verwendet einen hierarchischen Ansatz zur Verwaltung von Replikatgruppen. In Abbildung 6.8 sind zwei Replikatgruppen zu sehen. Oben im Bild ist eine minimale Replikatgruppe dargestellt, die lediglich ein Replikat beinhaltet. Jede Replikatgruppe verfügt über eine Objektgruppenreferenz. Die Gruppenreferenz ist eine persistente Objektreferenz, die von Clients zur Adressierung einer Replikatgruppe verwendet wird. Die einzelnen Elemente einer Replikatgruppe besitzen individuelle Referenzen, sogenannte Replikatreferenzen. Eine Replikatreferenz ist eine transiente Objektreferenz, mit deren Hilfe ein spezielles Replikat adressiert wird. Replikatreferenzen sind ausschließlich dem CORBA-Ablaufsystem bekannt. Weder der Servant eines Replikats noch seine Clients haben Kenntnis über die Replikatreferenz. Objektgruppenreferenzen und ihre zugehörigen Replikatreferenzen werden im Implementation Repository verwaltet. Die minimale Replikatgruppe aus Abbildung 6.8 besteht aus einer Objektgruppenreferenz und einem Replikat bzw. seiner Replikatreferenz. Clients die erstmalig auf eine Replikatgruppe zugreifen verwenden dazu die Objektgruppenreferenz, die auf das Implementation Repository verweist. Entsprechend dem Mechanismus der Anfrageumleitung erhalten Clients dann eine Replikatreferenz, mit der sie direkt mit einem bestimmten Replikat kommunizieren können. Der Mechanismus der Anfrageumleitung ist für Clients transparent, d.h. sie erlangen keine Kenntnis über die Replikatreferenz. Bei einer Migration oder Replikation ändert sich, wie im vorangegangenen Abschnitt dargestellt, nur die Replikatreferenz. Dies gewährleistet die Migrations und die Replikationstransparenz.

Mit dem hier vorgestellten Ansatz kann auch die statische Bindung realisiert werden. Im unteren Teil von Abbildung 6.8 ist eine Replikatgruppe mit drei Replikaten dargestellt. Diese unterscheidet sich von der zuvor beschriebenen minimalen Replikatgruppe durch eine Hierarchie sogenannter virtueller Objektreferenzen, die als Bin-

Abbildung 6.8 Schematische Darstellung der statischen Bindung



deglied zwischen der Objektgruppenreferenz und den Replikaten mit ihren Replikatreferenzen dienen. Virtuelle Objektreferenzen sind transiente Referenzen, die wiederum auf das Implementation Repository verweisen und ausschließlich dem CORBA-Ablaufsystem bekannt sind. Sie sind hierarchisch als eine Art Binärbaum strukturiert, der im folgenden als Bindungsbaum bezeichnet wird. An Gruppenreferenzen und virtuelle Referenzen sind entweder zwei virtuelle Referenzen oder eine Replikatreferenz gebunden. Durch die Übertragung des Konzepts der Bindungsbäume auf den Mechanismus der Anfrageumleitung kann die statische Bindung realisiert werden. Ursprünglich speichert die Laufzeitumgebung des Clients lediglich die Objektgruppenreferenz, um diese im Falle einer ungültig gewordenen Replikatreferenz wiederverwenden zu können. Erhält die Laufzeitumgebung durch eine `LocationForward`-Exception eine neue transiente Objektreferenz, so ersetzt sie die momentan benutzte durch die neue Objektreferenz. Der CORBA-Standard wird nun dahingehend erweitert, daß die Laufzeitumgebung eines Clients neben der Objektgruppenreferenz auch alle Objektreferenzen speichert, die sie durch `LocationForward`-Exceptions erhält. Die Objektreferenzen sind entsprechend ihrer Empfangsreihenfolge sortiert. Wird ei-

ne transiente Objektreferenz ungültig, so ersetzt sie die Laufzeitumgebung durch ihre Vorgängerreferenz. Das Implementation Repository realisiert die statische Bindung eines Clients an ein Replikat durch eine Reihe von `LocationForward`-Exceptions. Die dabei übermittelten Objektreferenzen entsprechen einem Pfad im Bindungsbaum, der von der Objektgruppenreferenz über beliebig viele virtuelle Objektreferenzen, bis hin zu einer Replikatreferenz führt. In der Laufzeitumgebung des Clients ist seine Bindung an ein bestimmtes Replikat dann durch eine geordnete Liste von Referenzen repräsentiert.

Bei der Migration eines Replikats wird im Bindungsbaum lediglich die Replikatreferenz des betreffenden Replikats durch eine neue Referenz ersetzt. Bei allen Clients des Replikats wird daraufhin die aktuell genutzte Replikatreferenz ungültig. Die Laufzeitumgebung ersetzt diese durch ihre Vorgängerreferenz. Dies ist entweder die Objektgruppenreferenz oder eine virtuelle Objektreferenz, die beide auf das Implementation Repository verweisen. Das Implementation Repository übermittelt dem Client schließlich die neue Replikatreferenz.

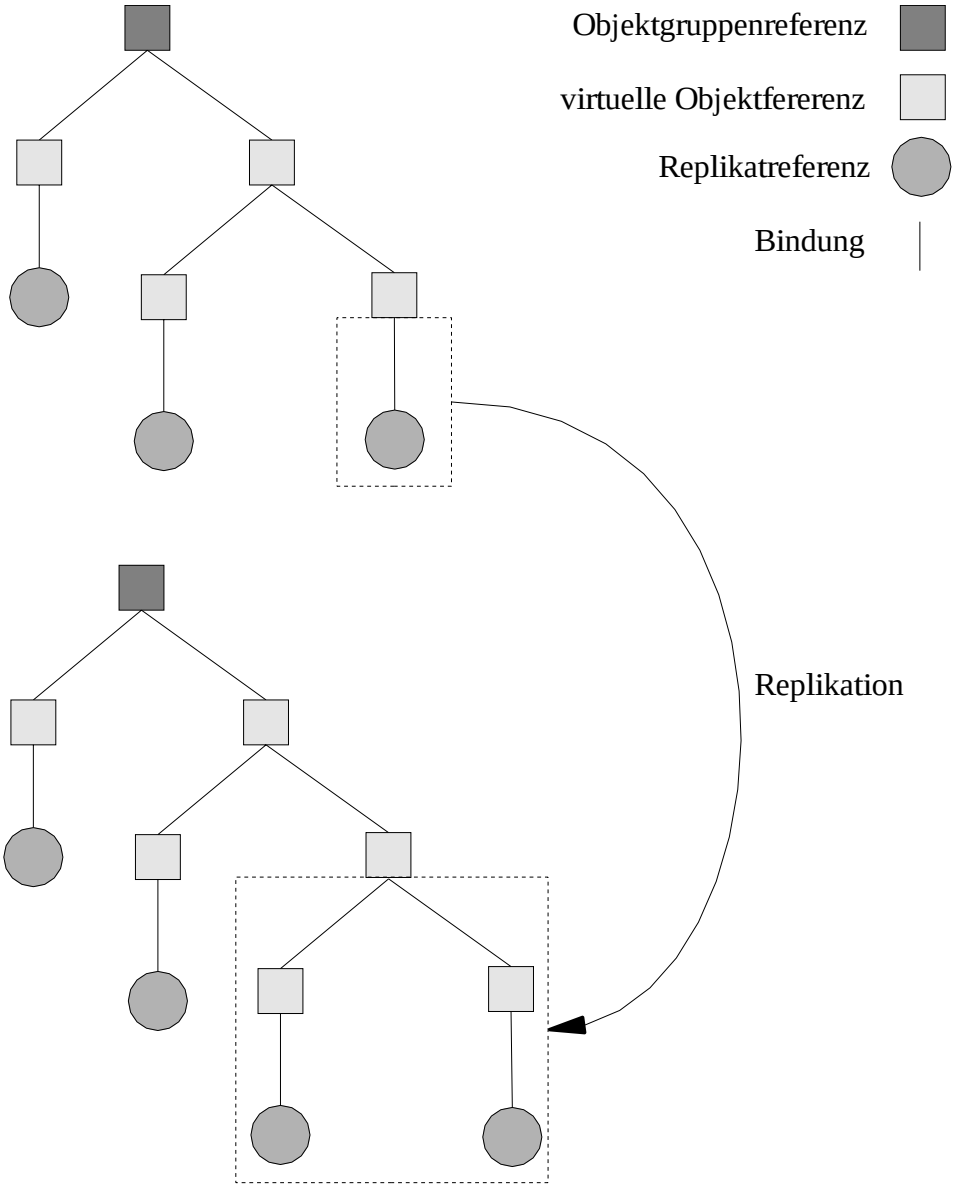
Die Realisierung der Replikation ähnelt stark der Vorgehensweise bei der Migration, erfordert aber größere Änderungen am Bindungsbaum. Abbildung 6.9 zeigt die Veränderung des Bindungsbaumes. Bei der Replikation ensteht ein neues Replikat, das eine eigene Replikatreferenz erhält. Die Replikatreferenz des ursprünglichen Replikats wird ebenfalls durch eine neue Referenz ersetzt. Zur Aufrechterhaltung der Integrität des Bindungsbaumes erzeugt das Implementation Repository zwei neue virtuelle Objektreferenzen, die zwischen den beiden Replikaten und der frei gewordenen virtuellen Objektreferenz eingefügt werden. Die Clients des ursprünglichen Replikats ersetzen die ungültige Replikatreferenz durch ihre Vorgängerreferenz, die als Objektgruppenreferenz oder als virtuelle Objektreferenz wiederum auf das Implementation Repository verweist. Das Implementation Repository teilt die Clients des ursprünglichen Replikats entsprechend der Weisung der Lastbewertung unter den beiden neuen Replikaten auf, indem es ihnen die neuen Replikatreferenzen übermittelt. Die Aufteilung der Clients auf die beiden Replikate bestimmt die Lastbewertung anhand des Verteilungswertes aus Kapitel 5.3.2.

Die Verwendung von Bindungsbäumen zur Realisierung von Replikatgruppen gewährleistet, daß aufeinanderfolgende Anfragen eines Clients stets an dasselbe Replikat gerichtet sind. Darüber hinaus ist bei der Lastverschiebung sichergestellt, daß die Clients des Quellreplikats ausschließlich an die entsprechenden Zielreplikate gebunden werden. Beide Eigenschaften zusammen gewährleisten nach Definition 4.14 die statische Bindung.

6.3.5 Verteilung der Anfragelast innerhalb einer Replikatgruppe

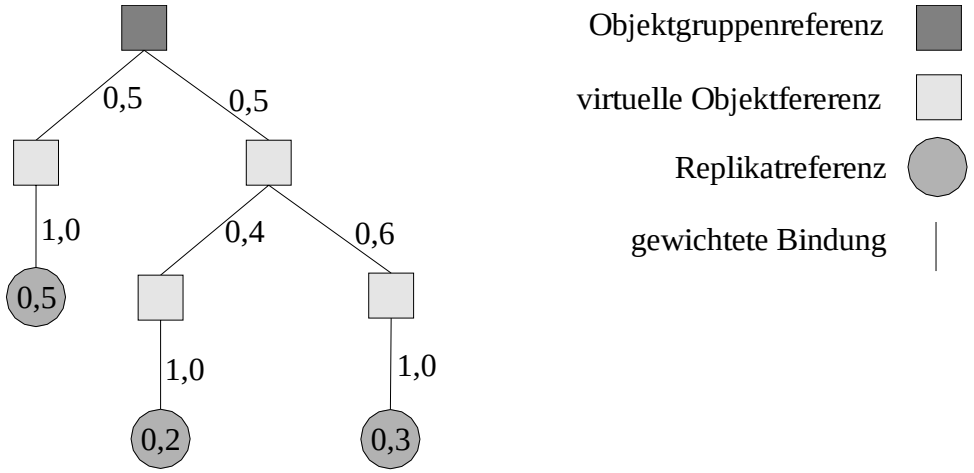
Bei der Replikation wird die Anfragelast eines Objekts auf zwei Replikate verteilt. Die Lastverteilung erreicht dies, indem sie die Clients eines Objekts in einem bestimmten Verhältnis zwischen seinen Replikaten aufteilt. Das Verteilungsverhältnis gibt die Lastbewertung durch den Verteilungswert vor. Betrachtet man nun eine Replikatgrup-

Abbildung 6.9 Statische Bindung und Replikation



pe, die durch mehrfache Replikation eines Ursprungsobjekts entstanden ist, so muß gewährleistet sein, daß die Anfragelast der Replikatgruppe entsprechend aller vorgegebenen Verteilungswerte zwischen den einzelnen Replikaten aufgeteilt wird. Diese Forderung hat maßgeblichen Einfluß auf die Bindung von Clients an Replikate durch die letztendlich die Verteilung der Anfragelast bestimmt ist. Für Clients, die ein Objekt bzw. eine Replikatgruppe von Beginn an nutzen ist dies unkritisch, da die statische Bindung die korrekte Verteilung der Anfragelast sicherstellt. Clients die erst später hinzukommen, müssen so zwischen den einzelnen Replikaten aufgeteilt werden, daß die korrekte Verteilung der Anfragelast innerhalb der Replikatgruppe weiter-

Abbildung 6.10 Bindung von Clients an Replikate

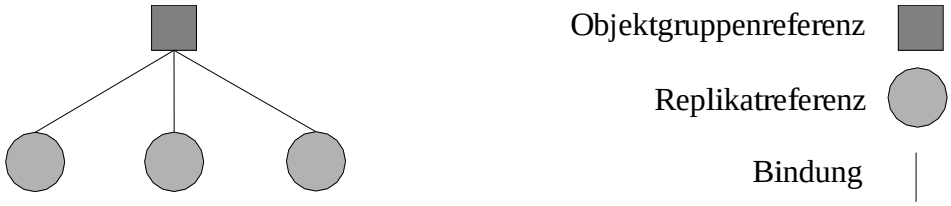


hin gewährleistet ist. Abbildung 6.10 verdeutlicht diesen Zusammenhang. Das Bild zeigt den Bindungsbaum einer Replikatgruppe, bei dem die Bindungen zwischen den einzelnen Objektreferenzen gewichtet sind. Das Gewicht einer Bindung ergibt sich aus dem Verteilungsverhältnis das bei der Entstehung der Bindung bzw. der entsprechenden Replikation zugrundegelegt wurde. Bindungen zu einer Replikatreferenz sind stets mit Eins gewichtet. Multipliziert man nun alle Gewichte auf dem Pfad von der Objektgruppenreferenz zu einer Replikatreferenz, so erhält man den Anteil der gesamten Anfragelast, den das jeweilige Replikat trägt. Clients die neu hinzukommen, müssen entsprechend dieser Anteile statistisch auf die Replikate verteilt werden.

6.3.6 Vorteile der statischen gegenüber der dynamischen Bindung

Die statische und die dynamische Bindung wurden bereits in Kapitel 4.3.6 miteinander vergleichen, wobei sich herausgestellt hat, daß die statische Bindung bei Objekten mit redundantem Zustand wesentliche Vorteile gegenüber der dynamischen Bindung bietet. An dieser Stelle können nun die beiden Bindungsarten hinsichtlich technischer Gesichtspunkte gegenübergestellt werden. Im Gegensatz zur statischen Bindung wäre bei der Realisierung der dynamischen Bindung keine hierarchische Datenstruktur nötig. Replikatgruppen könnten, wie in Abbildung 6.11 dargestellt, als flache Datenstruktur realisiert werden. Alle Replikatreferenzen einer Replikatgruppe wären dann direkt an die Objektgruppenreferenz gebunden. Der Hauptvorteil dieser Vorgehensweise läge darin, daß Bindungen zwischen Clients und Replikaten jederzeit gelöst und wiederhergestellt werden könnten. Damit bestünde die Möglichkeit die Arbeitslast innerhalb einer Replikatgruppe umzuverteilen, ohne daß dazu eine Replikation nötig wäre. Der Nachteil der dynamischen Bindung besteht jedoch darin, daß sie keine Gruppierung von Clients und deren Bindungen zuläßt, wie es mit Hilfe der virtuellen Objektreferenzen möglich ist. Damit muß die Lastverteilung bei jeder Aktion alle

Abbildung 6.11 Schematische Darstellung der dynamischen Bindung



Bindungen zwischen Clients und Replikaten lösen. Bei der statischen Bindung hingegen müssen nur die Bindungen der betroffenen Replikate gelöst werden. Das Lösen von Bindungen beeinflußt wiederum das Laufzeitverhalten der betroffenen Clients, da deren Anfragen dann über das Implementation Repository zu den Zielreplikaten umgeleitet werden müssen. Damit steigen die Kosten der Lastverteilung bei der dynamisch Bindung mit zunehmender Größe der Replikatgruppen. Bei der statischen Bindung hingegen sind die Kosten der Lastverteilung konstant. Die Anzahl der zu lösenden Bindungen ist insbesondere bei der Umverteilung von Clients innerhalb einer Replikatgruppe kritisch, da die Kosten Lastverteilung in diesem Fall ausschließlich durch die Kosten der Anfrageumleitung bestimmt sind. Deshalb kann man davon ausgehen, daß die Umverteilung von Clients nur für kleine Replikatgruppen praktikabel ist. Die Möglichkeit der Umverteilung von Clients wäre jedoch der Hauptvorteil der dynamischen Bindung, was wiederum die Entscheidung aus Kapitel 4.3.6 bestätigt, auf die Anwendung der dynamischen Bindung zu verzichten.

6.4 Zusammenfassung

In diesem Kapitel wurde das Lastverwaltungssystem LMC vorgestellt. LMC erfüllt alle anfangs gestellten Anforderungen hinsichtlich der Qualität der Implementierung. Zur Integration der Lastverwaltung in das CORBA-Programmiermodell waren geringfügige Erweiterungen des CORBA-Standards nötig. Diese betreffen insbesondere die Verwaltung des Objekt-Lebenszyklus und die Realisierung der statischen Bindung. Alle Erweiterungen des Standards sind konform zum CORBA-Objektmodell und könnten problemlos in zukünftige Versionen des CORBA-Standards aufgenommen werden. Die nahtlose Integration der Lastverwaltung in das Programmiermodell von CORBA und die Transparenz von Lasterfassung und Lastverteilung gewährleisten, daß der Mehraufwand für den Entwickler so gering als möglich ist. Die Verwendung allgemein anerkannter Standards und die grundlegende Architektur von LMC stellen die Performanz und die Skalierbarkeit der Lastverwaltung sicher.

Im weiteren Verlauf des Kapitel wurden die wichtigsten technischen Aspekte der Implementierung vorgestellt. Besondere Aufmerksamkeit erhielt dabei das Konzept der Bindungsbäume, mit deren Hilfe sowohl die statische Bindung als auch die Verteilung der Anfragelast innerhalb einer Replikatgruppe realisiert werden kann. Das Kapitel

schließt mit einem Vergleich von statischer und dynamischer Bindung. Dabei hat sich gezeigt, daß die statische Bindung nicht nur in konzeptioneller, sondern auch in technischer Hinsicht vorzuziehen ist.

7. Evaluierung des Lastverwaltungskonzepts

Ziel dieses Kapitels ist es, die Anwendbarkeit und den Nutzen des in dieser Arbeit vorgestellten Lastverwaltungskonzepts zu überprüfen. Als Grundlage hierfür dient das Lastverwaltungssystem LMC, das Gegenstand des vorangegangenen Kapitels war. Zuerst werden die Kosten des Lastverwaltungssystems quantitativ betrachtet, um daraus generelle Aussagen über seine Anwendbarkeit ableiten zu können. Anschließend folgen Untersuchung in einer Testumgebung und eine Fallstudie aus dem Bereich der medizinischen Bildverarbeitung. Diese sollen den Nutzen der Lastverwaltung und des zugrundeliegenden Konzepts für konkrete Anwendungen demonstrieren.

7.1 Bewertung von Lastverwaltungssystemen

Eine objektive und eindeutige Bewertung von Lastverwaltungssystemen ist in der Regel sehr schwierig, da der Nutzen der Lastverwaltung sowohl von der konkreten Lastsituation als auch von der Beschaffenheit der betrachteten Anwendungen abhängt. Einerseits kann der Nutzen der Lastverwaltung beliebig groß werden, wenn das Lastungleichgewicht groß genug ist. Andererseits können auch Lastsituationen auftreten, die den Nutzen der Lastverwaltung gänzlich zunichte machen. Dies ist insbesondere dann der Fall, wenn sich die Lastsituation so schnell verändert, daß die Lastverwaltung darauf nicht mehr geeignet reagieren kann. Schließlich hängt der Nutzen der Lastverwaltung auch von der Beschaffenheit der betrachteten Anwendungen ab. Anwendungen die eine feingranulare Lastverteilung erlauben, vergrößern den Spielraum der Lastverwaltung und damit auch deren potentiellen Nutzen. So sind die Einflußmöglichkeiten der Lastverwaltung bei zustandslosen Objekten und bei Objekten mit redundantem Zustand wesentlich größer als bei zustandsbehafteten Objekten, da in diesem Fall neben der Migration auch die Replikation anwendbar ist.

Die Bewertung des Lastverwaltungssystems LMC und seines zugrundeliegenden Lastverwaltungskonzepts erfolgt in drei Schritten. Zuerst werden die Kosten der Lastverwaltung und ihrer einzelnen Komponenten ermittelt, um ihre generelle Anwendbarkeit in speziellen Umgebungen abschätzen zu können. Daran schließen einige Un-

tersuchungen in einer Testumgebung an. Diese ermöglichen es, spezielle Teilaspekte der Lastverwaltung in einer frei parametrisierbaren Umgebung zu testen. Schließlich folgt eine Fallstudie mit einer realen Anwendung aus dem Bereich der medizinischen Bildverarbeitung. Diese soll den Nutzen der Lastverwaltung in einer realen Umgebung demonstrieren.

7.2 Kosten des Lastverwaltungssystems

Die Anwendbarkeit und der Nutzen eines Lastverwaltungssystems in einer konkreten Umgebung hängt stark von den Kosten der Lastverwaltung ab. Deshalb werden hier die Kosten des Lastverwaltungssystems LMC und seiner einzelnen Komponenten betrachtet. Tabelle 7.1 zeigt die einzelnen Komponenten von LMC und den Zeitbedarf für die Erledigung jeweils typischer Aufgabenstellungen. Die Messungen wurden

Tabelle 7.1 Die Kosten des Lastverwaltungssystems

Komponente	Aufgabe	Zeitbedarf [ms]
Lasterfassung	Erfassung aller Objektlasten	9
	Erfassung aller Rechnerlasten	44
Lastbewertung	Bewertung aller fiktiven Lastverteilungsaktionen	28
Lastverteilung	Secure Shell (SSH) starten	782
	Server−Prozeß erzeugen (ohne SSH)	1188
	Server−Prozeß anmelden	49
	Objekt erzeugen	192

in einer Testumgebung durchgeführt, die aus sechs vernetzten Arbeitsplatzrechnern besteht, die jeweils vier Objekte der im nächsten Abschnitt beschriebenen Testanwendung ausführen.[1] Die dargestellten Meßergebnisse sind vor dem Hintergrund zu sehen, daß ein Lastverwaltungszyklus bestehend aus Lasterfassung und Lastbewertung in der Praxis in Zeitintervallen von ein bis zwei Minuten durchlaufen wird. Lastverteilung findet nur dann statt, wenn die Lastbewertung eine geeignete Lastverteilungsaktion identifiziert hat.

Bei der Lasterfassung fällt auf, daß der Zeitbedarf für die Erfassung der Rechnerlastvektoren wesentlich größer ist als der für die Ermittlung der Objektlastvektoren. Der Grund dafür ist in der Art und Weise der Lastmessung zu finden. Ein Portable Object Adapter aktualisiert die Objektlastvektoren seiner Objekte fortlaufend, während die Werte für die Rechnerlastvektoren von den SNMP-Agenten erst auf Anfrage gemessen werden. Deshalb ist der Zeitbedarf für die Ermittlung der Objektlasten geringer

[1] Die Testumgebung umfaßt vier Sun Ultra 10 und zwei PCs mit AMD Athlon 900 MHz. Als Verbindungsnetzwerk kommt ein Fast Ethernet mit 100 MBit/s zum Einsatz.

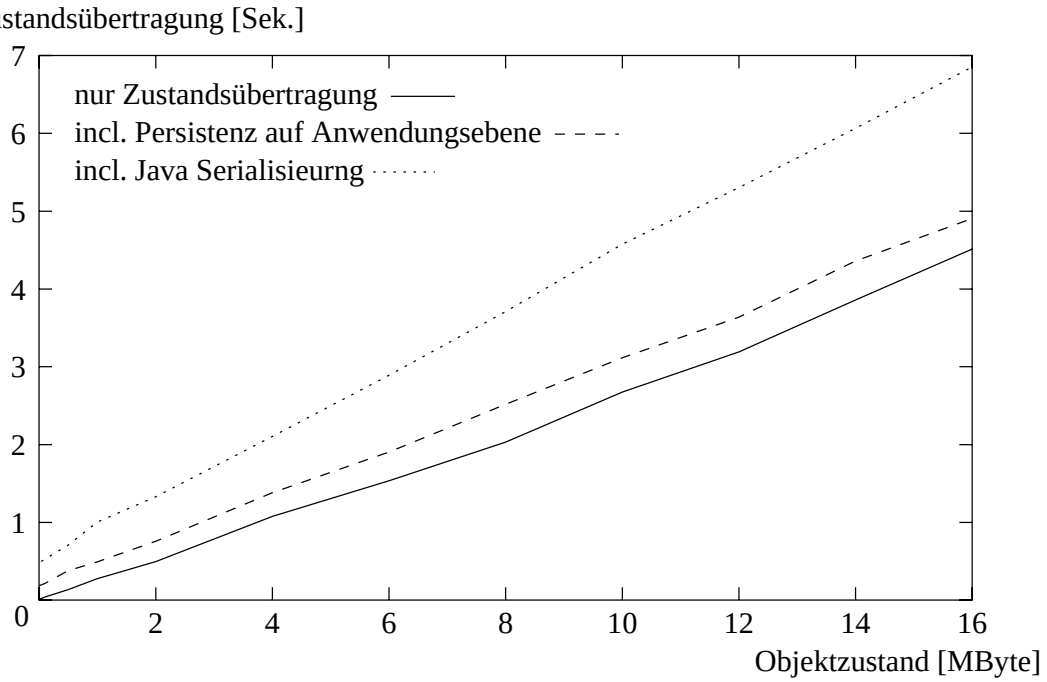
als der für die Erfassung der Rechnerlasten. Das Lastverwaltungssystem LMC betreibt die Lasterfassung größtenteils nebenläufig, so daß der Zeitbedarf kaum von der Anzahl der betrachteten Rechner und Objekte abhängt.

Anders verhält es sich bei der Lastbewertung, deren Zeitbedarf, wie in Kapitel 5.3 dargestellt wurde, sowohl durch die Anzahl der potentiellen Zielrechner als auch durch die Zahl der Objekte bestimmt ist. Absolut gesehen ist der zeitliche Aufwand für die Lastbewertung jedoch sehr gering.

Bei der Lastverteilung wird mit Hilfe der Secure Shell (SSH) [3] ein Server-Prozeß auf einem Zielrechner erzeugt. Der große Zeitbedarf für das Erzeugen des Server-Prozesses ist weniger auf den Umfang der Testanwendung als vielmehr auf die Größe der virtuellen Maschine von Java zurückzuführen. Das Anmelden des Server-Prozesses bei der Lastverwaltung und das anschließende Erzeugen eines Objekts sind im Vergleich zu den ersten beiden Arbeitsschritten wesentlich weniger zeitintensiv.

Der Zeitbedarf für die Zustandsübertragung muß gesondert betrachtet werden, da er maßgeblich von der Größe des Zustandes abhängt. Dieser Zusammenhang ist in Abbildung 7.1 dargestellt. Dabei wird zusätzlich zum Zeitbedarf für die Übertragung

Abbildung 7.1 Zeitbedarf für die Zustandsübertragung in Abhängigkeit von der Größe des Zustandes



des Zustandes auch der Aufwand für die Extraktion des Zustandes berücksichtigt. Als Persistenzmechanismen kommen die Persistenz auf Anwendungsebene und die Objektserialisierung von Java zum Einsatz. Der Vergleich mit dem Zeitbedarf für die rei-

ne Zustandsübertragung zeigt, daß die Kosten bei der Objektserialisierung wesentlich größer sind als bei der Persistenz auf Anwendungsebene.

Diese Untersuchungen zeigen, daß bei der Lasterfassung und Lastbewertung nur sehr geringe Kosten anfallen. Damit können diese Komponenten problemlos in größeren Umgebungen eingesetzt werden. Die Lastverteilung hingegen verursacht hohe Kosten, die jedoch nicht von der Größe des betrachteten verteilten Systems abhängen. Anhand dieser Untersuchungen kann nun der Nutzen der Lastverteilung für konkrete Anwendungen abgeschätzt werden.

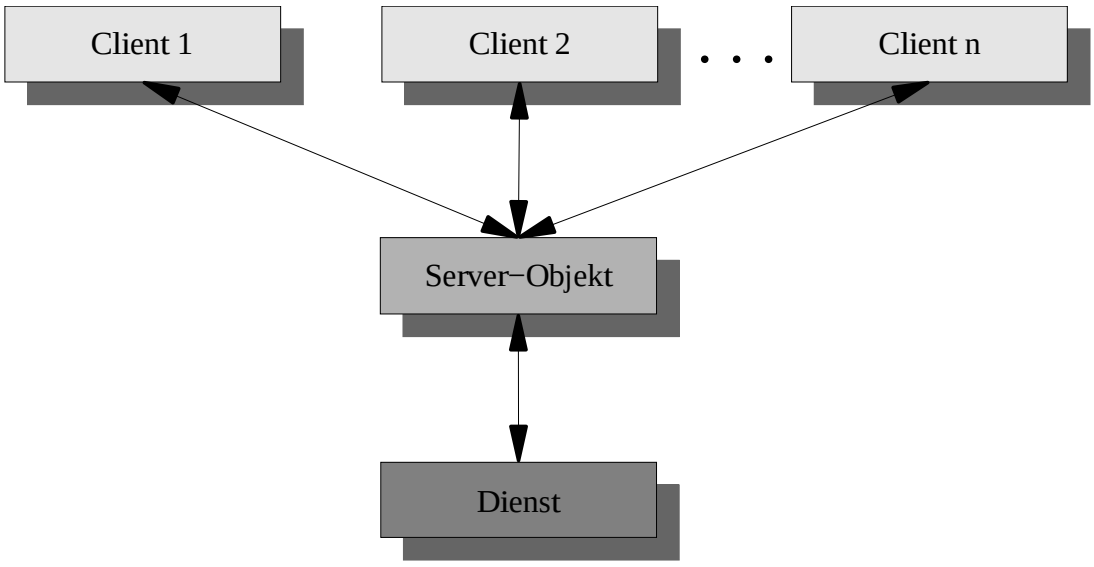
7.3 Die Testanwendung

Im folgenden wird eine Testanwendung vorgestellt, mit deren Hilfe man bestimmte Teilaspekte der Lastverwaltung innerhalb einer überschaubaren und gut kontrollierbaren Umgebung betrachten kann. Die nachfolgenden Abschnitte sollen aufzeigen, wie sich die Lastverwaltung bei konkreten Lastsituationen verhält. Damit kann überprüft werden, ob das Lastverwaltungssystem LMC prinzipiell in der Lage ist, die Lastbalancierung in einem verteilten System zu gewährleisten.

7.3.1 Aufbau der Testanwendung

Die hier vorgestellte Testanwendung ist 3-schichtig aufgebaut. Wie in Abbildung 7.2 zu sehen ist, findet man auf der obersten Ebene eine beliebig große Anzahl von Clients, die ständig Anfragen an ein Server-Objekt auf der mittleren Ebene stellen. Jeder Cli-

Abbildung 7.2 Schematische Darstellung der Testanwendung



ent ermittelt dabei seine Anfragerate, d.h. die Anzahl der Anfragen, die vom Server-Objekt pro Zeiteinheit beantwortet wurden. Zur Beantwortung einer Anfrage benötigt das Server-Objekt spezifische Daten, die es von einem externen Dienst bezieht, der auf der untersten Ebene angesiedelt ist. Mit Hilfe dieser Daten kann schließlich das Ergebnis der Anfrage berechnet werden. Somit teilt sich die Arbeit des Servers in die beiden Phasen Berechnung und Kommunikation. Die Dauer der einzelnen Phasen kann dem Server-Objekt über einen Parameter vorgegeben werden. Damit ist es möglich, das Verhältnis zwischen Kommunikation und Berechnung frei zu bestimmen. Das Server-Objekt ist zustandslos und damit sowohl migrierbar als auch replizierbar.

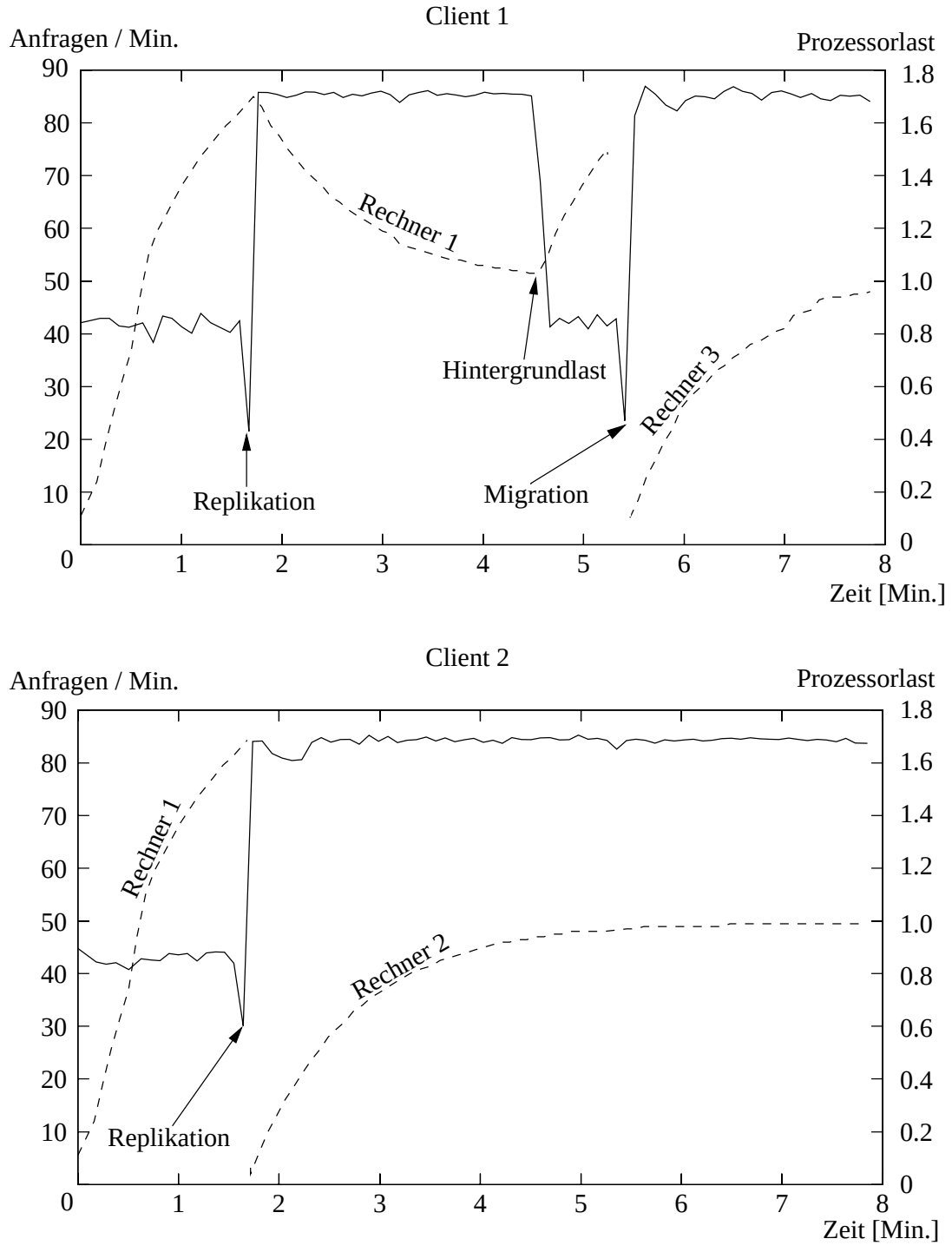
Für die Ausführung des Server-Objekts stehen drei homogene Arbeitsplatzrechner[2] zur Verfügung. Die übrigen Komponenten der Testanwendung, d.h. die Clients und der externe Dienst, laufen auf anderen Rechnern, um jede Beeinflussung des Server-Objekts auszuschließen. Mit Hilfe eines Lastgenerators kann auf jedem der drei Arbeitsplatzrechner eine definierte Prozessorlast erzeugt werden, um den Einfluß von Hintergrundlast zu simulieren.

7.3.2 Meßergebnisse und Bewertung

Die erste Messung soll demonstrieren, wie sich die Lastverwaltung an eine veränderte Lastsituation anpaßt. Dabei kommen zwei Clients zum Einsatz, die das Server-Objekt ständig anfragen. In Abbildung 7.3 sind zwei Diagramme zu sehen die jeweils die Anfragerate eines der beiden Clients in Abhängigkeit von der Zeit darstellen. Zusätzlich ist in jedem Diagramm die Prozessorlast des Rechners eingetragen, der das Replikat ausführt, an welches der jeweilige Client gebunden ist. Die Prozessorlast wird mit Hilfe des `avenrun`-Wertes des Betriebssystems ermittelt, der die Last, wie in Kapitel 5.4.1 beschrieben, geglättet und zeitverzögert wiedergibt. Im folgenden wird die Prozessorlast mit dem `avenrun`-Wert gleichgesetzt. Die Anfragerate ist als durchgezogene und die Prozessorlast als gestrichelte Linie dargestellt. Über den Kurven für die Prozessorlast ist jeweils der Rechner angegeben, auf den sich diese Last bezieht. Zu Beginn der Messung erzeugt die Lastverwaltung das Server-Objekt auf Rechner 1, wobei die Auswahl an dieser Stelle beliebig ist, da die einzelnen Rechner homogen und unbelastet sind. Zu diesem Zeitpunkt umfaßt die Replikatgruppe des Server-Objekts nur ein Replikat, so daß beide Clients an dasselbe Replikat gebunden sind. Die Anfragerate der Clients pendelt sich rasch auf einem bestimmten Niveau ein. Aufgrund der Glättung des `avenrun`-Wertes steigt die Prozessorlast nur langsam an. Mit der steigenden Prozessorlast auf Rechner 1 erhöht sich auch das Lastungleichgewicht im Rechensystem. Deswegen entscheidet sich die Lastbewertung schließlich für eine Replikation des Server-Objekts. Eine Migration steht in diesem Moment nicht zur Debatte, da eine Verschiebung der Last auf einen anderen Rechner das Lastungleichgewicht nicht verringern würde. Die Replikation äußert sich in einem kurzzeitigen Abfall der Anfragerate beider Clients. Wegen der Homogenität der beteiligten Rechner und der fehlenden Hintergrundlast wird die Anfragelast des Server-Objekts gleichmäßig auf

[2] Als Arbeitsplatzrechner kommen drei Sun Ultra 10 zum Einsatz, die durch ein Fast Ethernet mit 100 MBit/s verbunden sind.

Abbildung 7.3 Anpassung der Lastverwaltung an eine veränderte Lastsituation



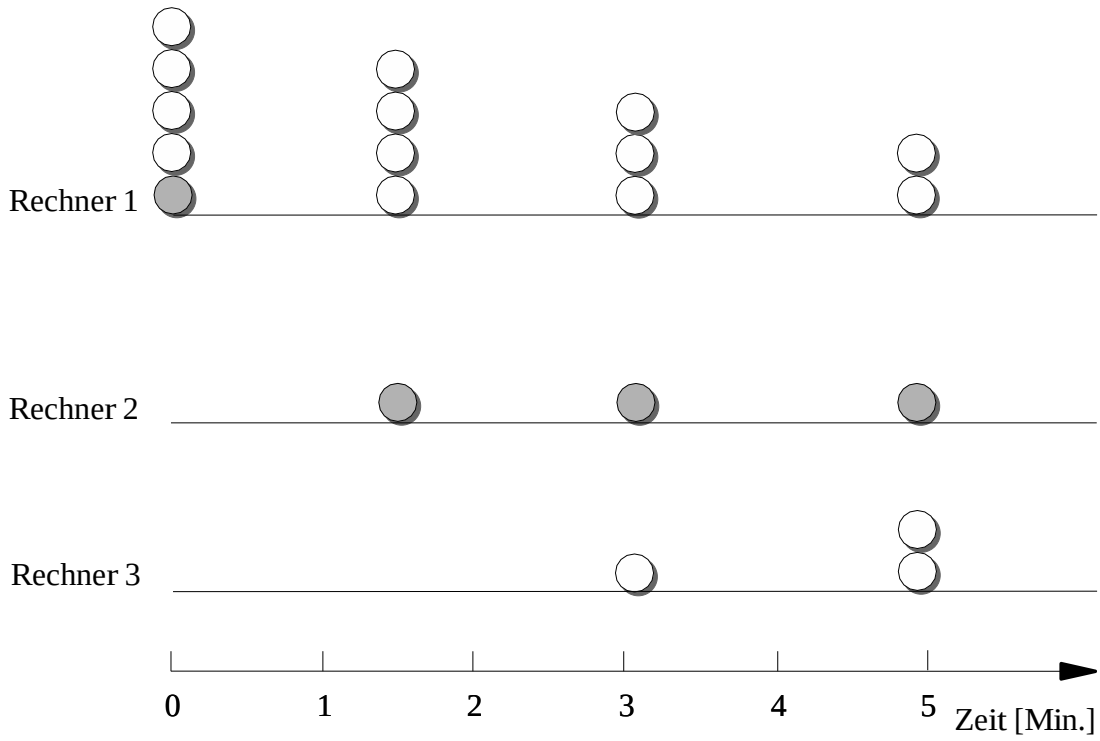
die beiden Replikate verteilt. Folglich bindet die Lastverteilung jeden Client an eines der beiden Replikate. Entsprechend verdoppelt sich durch die Replikation die Anfragerate der Clients. Die Last auf dem Quellrechner (Rechner 1) sinkt und die Last auf dem Zielrechner (Rechner 2) beginnt langsam anzusteigen. In dieser Situation ist ein gewisser Gleichgewichtszustand erreicht, da die Lastbewertung keine Aktion findet mit der sie das Lastungleichgewicht des Rechensystems weiter verringern könnte. Nach einer gewissen Zeit wird mit Hilfe des Lastgenerators Hintergrundlast auf Rechner 1 erzeugt. Daraufhin steigt die Prozessorlast auf Rechner 1 an und die Anfragerate von Client 1 sinkt. Die Last, die das Replikat auf diesem Rechner erzeugt, ist kleiner als der Schwellenwert Δ_R[3], der in Kapitel 5.3.2 beschrieben wurde. Deshalb kommt in dieser Situation eine Replikation nicht in Frage und die Lastbewertung entscheidet sich für eine Migration. Das Ziel dieser Migration ist Rechner 3, der bisher noch nicht genutzt wurde und somit noch völlig unbelastet ist. Als Ergebnis der Migration steigt die Lastkurve von Rechner 3 ausgehend von der Nullinie langsam an und die Anfragerate von Client 1 pendelt sich wieder auf ihr ursprüngliches Niveau ein.

In einem zweiten Versuch wird nun untersucht, wie die Lastverwaltung mit Objekten mit unterschiedlich hohen Ressourcenanforderungen umgeht. Zu diesem Zweck werden unter Umgehung der Initialplazierung fünf Instanzen des Server-Objekts auf einem Rechner erzeugt. Abbildung 7.4 zeigt die fünf Objekte und ihre ausführenden Rechner über der Zeit. Die vier Objekte, die als leere Kreise dargestellt sind, wurden so parametrisiert, daß sie zu gleichen Teilen rechnen und kommunizieren. Diese Objekte erzeugen folglich jeweils eine Prozessorlast von $0{,}5$. Das fünfte Objekt ist als gefüllter Kreis gezeichnet und betreibt keine Kommunikation. Damit verursacht dieses Objekt eine Prozessorlast von Eins. Das Bild zeigt, daß die Lastverwaltung zuerst das Objekt mit der größten Last auf einen anderen Rechner migriert, da diese Aktion das Lastungleichgewicht des Rechensystems am meisten verringert. Anschließend werden schrittweise zwei der anderen Objekte auf den noch ungenutzten Rechner verschoben. Im Endzustand ist damit die Prozessorlast auf allen beteiligten Rechnern gleich groß und das Lastungleichgewicht des Rechensystems ist ausgeglichen.

Diese Messungen zeigen die grundsätzliche Arbeitsweise des Lastverwaltungssystems LMC auf. Die Lastverwaltung gleicht das Lastungleichgewicht des Rechensystems aus und verbessert damit das Laufzeitverhalten der Testanwendung spürbar. Dies ist insbesondere auf die kombinierte Anwendung der Migration und der Replikation zurückzuführen. Während die Migration geeignet ist, um auf Hintergrundlast zu reagieren, ermöglicht es die Replikation, die Anfragelast eines Objekts auf mehrere Rechner zu verteilen. Nur das Zusammenspiel beider Mechanismen gewährleistet, daß die Lastverwaltung geeignet auf die vielfältigen Lastsituationen reagieren kann, die in verteilten objektorientierten Systemen auftreten. Darüber hinaus berücksichtigt die Lastverwaltung die spezifischen Ressourcenanforderungen der Anwendung und ihrer

[3]Für alle Messungen wird ein Schwellenwert Δ_R von Eins verwendet. Damit repliziert die Lastverwaltung nur Objekte, deren Prozessorlast größer als Eins ist. Dies ist insofern sinnvoll, da die Last die ein Client erzeugen kann maximal Eins ist. Damit wird verhindert, daß die Lastverwaltung Objekte repliziert, die nur einen Client haben.

Abbildung 7.4 Objekte mit unterschiedlich hohen Ressourcenanforderungen



einzelnen Objekte. Dadurch kann die Lastverwaltung zielgerichtet agieren, wodurch Fehlentscheidungen weitgehend vermieden werden können.

7.4 Eine Anwendung aus der medizinischen Bildverarbeitung

Im folgenden wird eine Anwendung aus dem Bereich der medizinischen Bildverarbeitung vorgestellt. Diese Anwendung dient als Grundlage für eine Reihe von Untersuchungen, welche die Anwendbarkeit und den Nutzen des Lastverwaltungssystems LMC bei realen Anwendungen demonstrieren sollen.

7.4.1 Aufbau der Anwendung

Bei der hier untersuchten Anwendung handelt es sich um die sogenannte Realignment-Komponente des Programmpakets SPM (Statistical Parametric Mapping) [146, 33]. SPM wird am Wellcome Department of Cognitive Neurology des University College London entwickelt, um die Aktivitäten des menschlichen Gehirns bei motorischen und kognitiven Experimenten analysieren zu können. Grundlage für diese Untersu-

chungen sind Bildsequenzen, die von Positronen-Emissions- oder Magnet-Resonanz-Tomographen stammen [82]. Das Programmpaket SPM dient der Aufbereitung und Verarbeitung dieser Bildsequenzen. Die Aufgabe der Realignment-Komponente ist es, die Bewegungen des untersuchten Patienten aus den Bildsequenzen herauszufiltern. Dazu müssen alle Bilder einer Sequenz mit dem ersten Bild verglichen und gegebenenfalls neu ausgerichtet werden.

Die Realignment-Anwendung ist 2-schichtig aufgebaut. Ein Client sendet eine oder mehrere Bildsequenzen an ein Server-Objekt. Dabei werden die Bilder jeder Sequenz einzeln an das Server-Objekt geschickt, das dann eine entsprechende Transformationsmatrix an den Client zurückliefert. Zur Bearbeitung der einzelnen Bilder einer Sequenz muß der Server diese mit dem jeweils ersten Bild der Sequenz vergleichen. Dazu verwendet er Referenzdaten, die der Client zur Verfügung stellt. Dieser errechnet die Referenzdaten auf Anfrage aus dem ersten Bild einer Sequenz und gegebenenfalls den Ergebnisdaten der Vorgängersequenz. Wenn das Server-Objekt ein Bild von einer bisher unbekannten Sequenz erhält, dann fragt es die zugehörigen Referenzdaten beim Client an. Der Server legt die Referenzdaten dann in einem Zwischenspeicher ab, um sie für weitere Bilder derselben Sequenz wiederverwenden zu können. Ist der Zwischenspeicher voll, dann entfernt der Server die Referenzdaten, die er die längste Zeit nicht mehr benötigt hat. Der Zustand des Server-Objekts besteht lediglich aus seinem Zwischenspeicher. Damit verfügt der Server über einen redundanten Zustand und ist, wie bereits in Kapitel 4.3.4 beschrieben wurde, sowohl migrierbar als auch replizierbar.

Das Programmpaket SPM wurde in der Programmiersprache C entwickelt und benutzt für die numerischen Berechnungen die Mathematik-Software MATLAB [28]. Die ursprüngliche Realignment-Anwendung berechnet alle Bilder sequentiell. Um die Abarbeitung der Bildsequenzen zu beschleunigen hat MAY [76] die Realignment-Anwendung aus dem Programmpaket SPM herausgelöst und in eine verteilte Anwendung transformiert. STAMATAKIS [127] machte die Realignment-Anwendung dem Java-basierten JacORB zugängig, indem er sie mit Hilfe des Java Native Interface (JNI) [134] gekapselt und dahingehend erweitert hat, daß sie mehrere Bilder parallel abarbeitet. Zu diesem Zweck werden im Client eine fest vorgegebene Anzahl von Ausführungsfäden (Threads) abgespalten, die den Server nebenläufig anfragen. Damit spiegelt die Anfragelast des Servers die Arbeitslast der gesamten Anwendung wieder und die Lastverwaltung kann geeignete Maßnahmen ergreifen, um die Arbeitslast auf das gesamte Rechensystem zu verteilen.

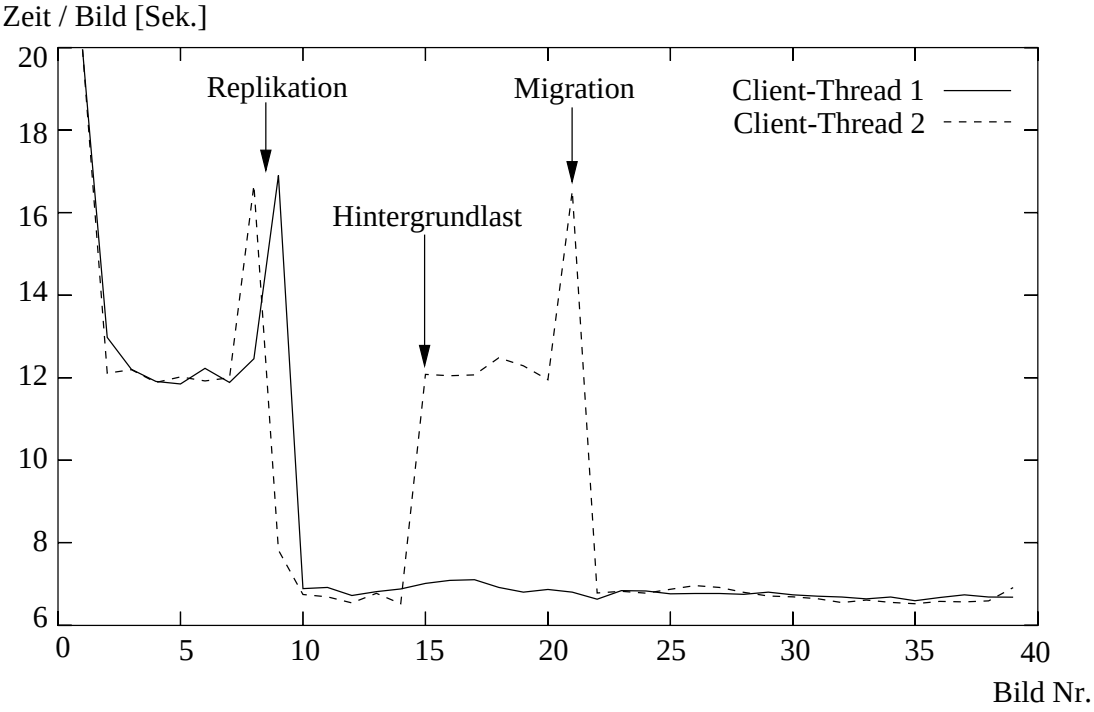
7.4.2 Meßergebnisse und Bewertung

Bei der ersten Messung mit der Realignment-Anwendung wird untersucht, wie die Lastverwaltung auf unterschiedliche Lastquellen reagiert. In Kapitel 4.2.2 wurde beschrieben, daß in verteilten objektorientierten Systemen zwischen Anfragelast und Hintergrundlast unterschieden werden muß. Die Hintergrundlast stammt von Anwendungen, die außerhalb der Kontrolle der Lastverwaltung liegen. Anfragelast hingegen wird von der lastverwalteten Anwendung selbst verursacht. Die Lastverwaltung kann

der Anfragelast durch die Replikation und die damit einhergehende Umverteilung von Anfragen entgegenwirken.

Die Testumgebung besteht aus drei Rechnern[4], von denen einer die Realignment-Anwendung ausführt. Die verbleibenden beiden Rechner werden zu Beginn nicht genutzt. Außerhalb der Testumgebung läuft ein Client mit zwei Threads, die mit Hilfe der Realignment-Anwendung Bildsequenzen berechnen. Abbildung 7.5 zeigt die Berechnungszeit für die einzelnen Bilder jedes Threads. Zur Bearbeitung des ersten

Abbildung 7.5 Umgang mit unterschiedlichen Lastquellen



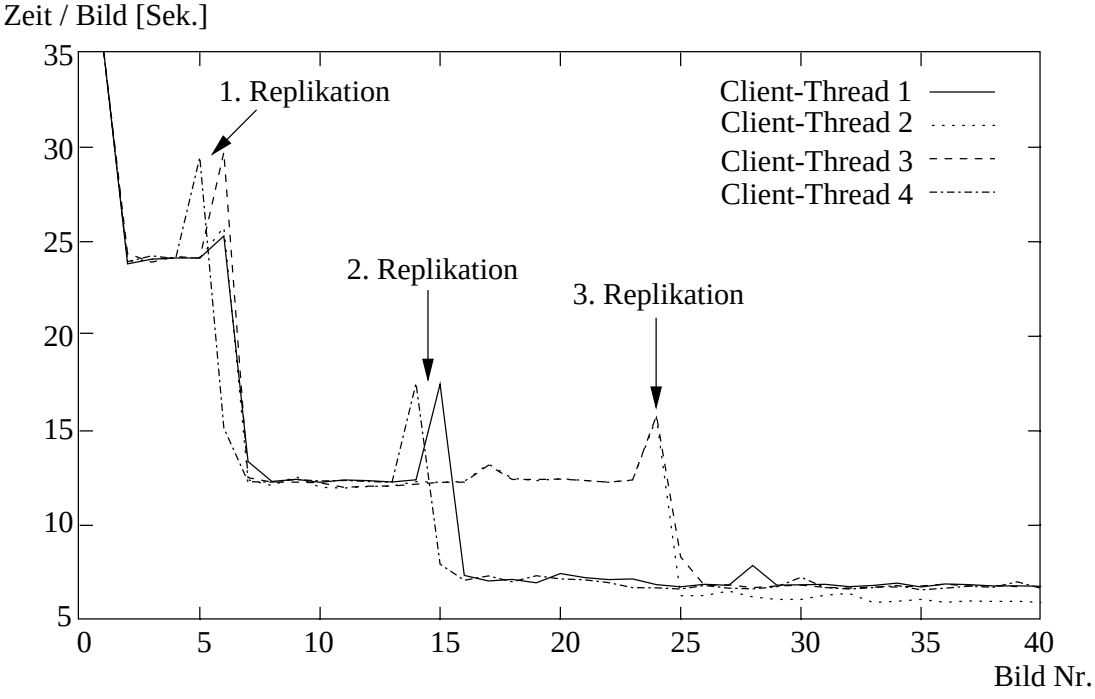
Bildes benötigt der Server die oben erwähnten Referenzdaten. Da der Client diese erst berechnen muß, ist der Zeitaufwand für das erste Bild sehr groß. Anschließend pendelt sich die Berechnungszeit bei beiden Client-Threads rasch auf einem bestimmten Niveau ein. Am Anfang benutzen beide Threads dasselbe Server-Objekt, was zu Überlast auf dem Server-Rechner führt. Deshalb entscheidet sich die Lastverwaltung für eine Replikation des Server-Objekts auf einen der freien Rechner. Eine Migration kommt in diesem Fall nicht in Frage, da die Verschiebung der Last auf einen anderen Rechner das Lastungleichgewicht des Rechensystems nicht verringern würde. Der Mehraufwand, der durch die Replikation entsteht, ist an einem sprunghaften Anstieg der Berechnungsdauer zu erkennen. Infolge der Replikation reduziert sich schließlich die Berechnungszeit wieder auf die Hälfte ihres vorherigen Wertes, da sich die Anfragelast nun gleichmäßig auf die beiden Replikate verteilt. Nach einer gewissen Zeit wird

[4]Die Testumgebung für die Realignment-Anwendung umfaßt mehrere Sun Ultra 10 Rechner. Als Verbindungsnetzwerk kommt ein Fast Ethernet mit 100 MBit/s zum Einsatz.

mit Hilfe des Lastgenerators Hintergrundlast auf einem der beiden Server-Rechner erzeugt. Der entsprechende Rechner ist dadurch überlastet, was daran zu sehen ist, daß die Berechnungszeit beim Client-Thread des betroffenen Replikats stark ansteigt. Um dem Lastungleichgewicht entgegenzuwirken, migriert die Lastverwaltung das Replikat auf den verbleibenden freien Rechner, woraufhin sich die Berechnungszeit des Client-Threads wieder verringert.

Eine weitere Messung mit der Realignment-Anwendung soll aufzeigen, daß die Lastverwaltung die Arbeitslast der Anwendung auch bei größeren Problemstellungen gleichmäßig auf das Rechensystem verteilt. Die verwendete Testumgebung besteht aus fünf Rechnern. Einer dieser Rechner führt die Realignment-Anwendung aus, die übrigen Rechner sind unbelastet. Außerhalb der Testumgebung läuft ein Client mit vier Threads. In Abbildung 7.6 ist wiederum die Berechnungszeit für die einzelnen Bilder jedes Threads zu sehen. Im Verlauf der Messung führt die Lastverwaltung

Abbildung 7.6 Mehrfache Anwendung der Replikation



drei Replikationen durch. Nach der ersten Replikation bedient jedes Replikat zwei Client-Threads, wodurch sich die Berechnungsdauer für die einzelnen Bilder halbiert. Bei der zweiten Replikation wird eines der beiden Replikate erneut repliziert. Infolgedessen halbiert sich die Berechnungszeit der betroffenen Client-Threads. Auch der Mehraufwand, der durch die Replikation verursacht wird, wirkt sich wegen der statischen Bindung nur auf diese beiden Threads aus. Nach der dritten Replikation verfügt schließlich jeder Client-Thread über ein eigenes Replikat des Realignment-Objekts. Zu diesem Zeitpunkt werden nur vier der fünf verfügbaren Rechner genutzt. Trotzdem

führt die Lastverwaltung keine weitere Replikation durch, da die Last der einzelnen Replikate kleiner ist als der Schwellenwert Δ_R. Damit verhindert die Lastbewertung, daß Objekte repliziert werden, die nur einen Client haben.

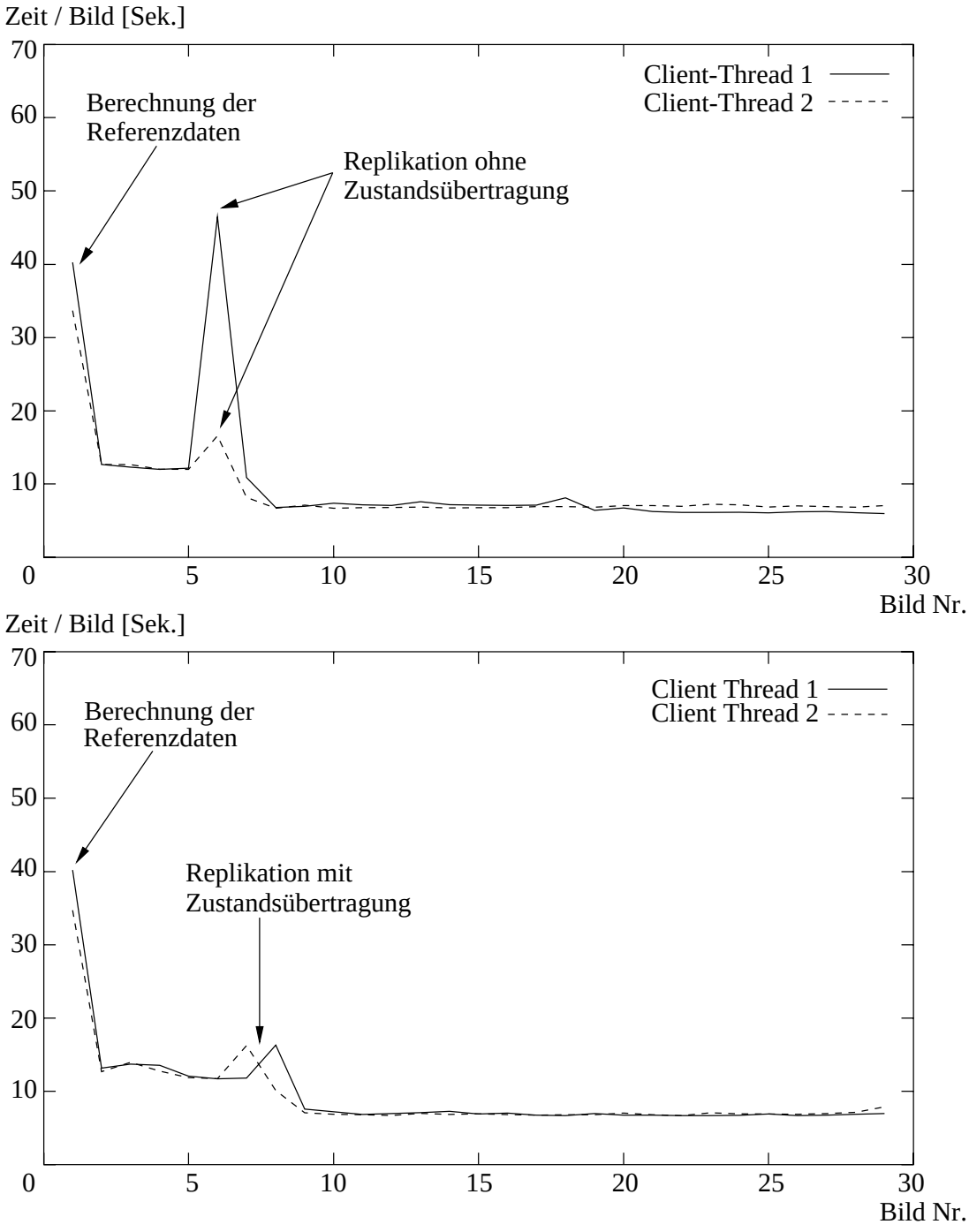
In den bisher durchgeführten Untersuchungen wurde bei der Lastverschiebung stets der Zustand des Quellobjekts zum Zielobjekt übertragen. Anhand einer weiteren Messung soll nun untersucht werden, inwiefern die Zustandsübertragung bei der Realignment-Anwendung von Vorteil ist. In diesem Fall umfaßt die Testumgebung zwei Rechner. Einer der beiden Rechner führt die Realignment-Anwendung aus, der andere Rechner ist unbelastet. Außerhalb der Testumgebung fragt ein Client mit zwei Threads das Realignment-Objekt an. In Abbildung 7.7 sind zwei Testläufe dargestellt. Beim oberen Testlauf wurde mit und beim unteren ohne Zustandsübertragung repliziert. Es ist zu sehen, daß die Replikation ohne Zustandsübertragung einen erheblichen Mehraufwand beim neu erzeugten Replikat verursacht. Für die Rekonstruktion seines Zustandes muß dieses Replikat die Referenzdaten beim Client anfragen. Da die Berechnung der Referenzdaten sehr zeitaufwendig ist, fällt der Mehraufwand entsprechend groß aus. Die Kosten für die Rekonstruktion des Zustandes übersteigen die für die Zustandsübertragung bei weitem. Deshalb ist die Zustandsübertragung bei der Realignment-Anwendung sehr vorteilhaft.

Diese Messungen zeigen, daß das Laufzeitverhalten der Realignment-Anwendung durch die Lastverwaltung erheblich verbessert werden kann. Die Lastverwaltung ist in der Lage, die Arbeitslast der Anwendung mit Hilfe der Replikation auf das gesamte Rechensystem zu verteilen und somit die Abarbeitung der Bildsequenzen und ihrer einzelnen Bilder zu beschleunigen. Durch die Migration kann die Lastverwaltung dem Auftreten von Hintergrundlast entgegenwirken, indem sie die betroffenen Replikate auf einen weniger stark belasteten Rechner verschiebt. Darüber hinaus wird deutlich, daß die gesonderte Betrachtung von Objekten mit redundantem Zustand sinnvoll und notwendig ist. Nur mit Hilfe der dadurch möglichen Optimierungen, wie der Zustandsübertragung und der statischen Bindung, können die Kosten der Lastverschiebung niedrig gehalten werden. Insgesamt gesehen sind die Kosten der Lastverwaltung in Relation zu ihrem Mehrwert sehr gering. Dies bestätigt, daß das Lastverwaltungssystem LMC auch für real existierende Anwendungen deutliche Vorteile bietet.

7.5 Zusammenfassung

Die Bewertung von Lastverwaltungssystemen ist schwierig, da der Nutzen der Lastverwaltung sowohl von der Lastsituation als auch von der betrachteten Anwendung abhängt. Deshalb wurden in diesem Kapitel zuerst die Kosten des Lastverwaltungssystems LMC untersucht, um seine Anwendbarkeit in konkreten Umgebungen abschätzen zu können. Dank der effizienten Implementierung des Lastverwaltungssystems fallen die Kosten für die Lasterfassung und die Lastbewertung sehr gering aus. Im Vergleich dazu sind die Kosten für die Lastverteilung hoch. Insbesondere das Starten und Erzeugen von Server-Prozessen ist sehr zeitaufwendig. In der Praxis hat sich jedoch herausgestellt, daß dieser Mehraufwand durch den Nutzen der Lastverteilung

Abbildung 7.7 Vergleich der Replikation ohne Zustandsübertragung (oben) und mit Zustandsübertragung (unten)



rasch aufgewogen wird. Im Anschluß an die Kostenbetrachtungen wurde eine Reihe von Messungen in einer Testumgebung durchgeführt, um die grundlegende Arbeitsweise des Lastverwaltungssystems LMC zu veranschaulichen. Dabei hat sich gezeigt, daß die Lastverwaltung sehr gut auf die unterschiedlichen Lastsituationen reagiert, die in verteilten objektorientierten Systemen auftreten. Als unentbehrlich hat sich dabei die Kombination unterschiedlicher Lastverteilungsmechanismen erwiesen. Mit Hilfe der Migration kann die Lastverwaltung auf Hintergrundlast reagieren, indem sie Objekte von belasteten auf weniger stark belastete Rechner verschiebt. Die Replikation wird eingesetzt, um die Anfragelast einer Anwendung und ihrer Objekte auf mehrere Rechner zu verteilen. Nur durch die Kombination beider Mechanismen kann das Potential der Lastverwaltung in verteilten objektorientierten Systemen voll ausgeschöpft werden. Darüber hinaus verdeutlichen diese Untersuchungen die Vorzüge des hier vorgestellten Lastbewertungsalgorithmus und seines zugrundeliegenden Lastmodells. Durch die Berücksichtigung der spezifischen Ressourcenanforderungen der Objekte kann die Lastverwaltung zielgerichtet agieren, was Fehlentscheidungen größtenteils ausschließt. Das Kapitel endet mit einer Fallstudie aus dem Bereich der medizinischen Bildverarbeitung. Dabei wurde die Realignment-Komponente des Programmpakets SPM der Lastverwaltung unterzogen. Die Untersuchungen mit der Realignment-Anwendung bestätigen, daß sich die sehr guten Ergebnisse, die mit der Testanwendung erzielt wurden, auch auf praxisrelevante Anwendungen übertragen lassen. Darüber hinaus zeigt die Realignment-Anwendung, daß die Betrachtung des redundanten Zustandes und die damit verbundenen Optimierungsmöglichkeiten die Wirksamkeit der Lastverteilung stark erhöhen. Insgesamt gesehen unterstreicht dieses Kapitel den Nutzen und die Anwendbarkeit des Lastverwaltungssystems LMC und die Tragfähigkeit des zugrundeliegenden Lastverwaltungskonzepts.

8. Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Ergebnisse der Arbeit zusammengefaßt und bewertet. Im Anschluß daran folgt ein Ausblick auf eine Reihe von interessanten Themengebieten, die für eine Fortführung dieser Arbeit geeignet wären.

8.1 Zusammenfassung

In dieser Arbeit wurde ein Konzept zur Lastverwaltung in verteilten objektorientierten Systemen entwickelt. Im Gegensatz zu früheren Ansätzen, die lediglich Lastverwaltungskonzepte anderer Programmiermodelle auf verteilte objektorientierte Systeme übertragen, wurde hier ein neues Konzept entwickelt, das an den speziellen Anforderungen und Möglichkeiten dieses Programmiermodells ausgerichtet ist.

Zu Beginn der Arbeit wurde ein Überblick über verteilte Systeme und deren Programmiermodelle gegeben. Das Hauptaugenmerk war dabei auf die technischen Merkmale der unterschiedlichen Programmiermodelle und ihre Einordnung in den historischen Kontext gerichtet. Das Spektrum der betrachteten Systeme reicht von einfachen nachrichtenorientierten Systemen über den entfernten Prozeduraufruf, bis hin zu verteilten objektorientierten Systemen. Dabei wurde ein einheitliches Begriffsgerüst geschaffen, das die Grundlage für die gesamte Arbeit bildet.

Die Problematik der Lastungleichheit und die daraus resultierende Notwendigkeit zur Lastverwaltung besteht bei allen verteilten Anwendungen, unabhängig von deren Programmiermodell. Im Lauf der Zeit haben sich deshalb schon einige Arbeiten mit dieser Thematik beschäftigt. Die Ergebnisse dieser Arbeiten können jedoch nicht auf verteilte objektorientierte Systeme übertragen werden, da sich diese sowohl in ihren Anforderungen als auch in ihren Möglichkeiten grundlegend von anderen Programmiermodellen unterscheiden. Für die Entwicklung eines neuen Lastverwaltungskonzepts mußte zuerst die reichhaltige Terminologie aus dem Bereich der Lastverwaltung vereinheitlicht und an die besonderen Gegebenheiten verteilter objektorientierter Systeme angepaßt werden. Zu diesem Zweck wurde ein Klassifikationsmodell für Lastverwaltungssysteme entwickelt, das in die Komponenten Lasterfassung, Lastbewertung

und Lastverteilung gegliedert ist. Dies ermöglichst es, die einzelnen Komponenten der Lastverwaltung getrennt voneinander zu betrachten, was die Komplexität des Klassifikationsmodells spürbar reduziert. Mit Hilfe des Klassifikationsmodells wurde dann aus den speziellen Eigenschaften verteilter objektorientierter Systeme ein Lastverwaltungskonzept entwickelt.

Wesentliche Neuerungen dieses Konzepts gegenüber anderen Ansätzen sind die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen, die Unterstützung zustandsbehafteter Objekte und die Transparenz der Lastverwaltung. Ein Vergleich mit bestehenden Lastverwaltungssystemen hat gezeigt, daß diese meist nur einen Teil der möglichen Mechanismen zur Lastverteilung nutzen. Im Gegensatz dazu verwendet das hier präsentierte Konzept sowohl die Initialplazierung, die Migration als auch die Replikation. In der Evaluierung des Lastverwaltungskonzepts wurde deutlich, daß erst die kombinierte Anwendung dieser Lastverteilungsmechanismen eine geeignete Behandlung der vielen unterschiedlichen Lastsituationen ermöglicht, die in verteilten objektorientierten Systemen auftreten können. Die Berücksichtigung von zustandslosen und zustandsbehafteten Objekten macht der Lastverwaltung das gesamte Spektrum der verteilten objektorientierten Anwendungen zugängig. Frühere Ansätze unterstützen lediglich zustandslose Objekte und schränken damit die Anwendbarkeit der Lastverwaltung stark ein. Ein weiterer Kernpunkt dieses Konzepts ist die Transparenz der Lastverwaltung. Dadurch fällt der Mehraufwand für den Entwickler wesentlich geringer aus als bei den bestehenden Lastverwaltungssystemen. Dies trägt letztendlich dazu bei, die Entwicklungskosten niedrig zu halten und die Akzeptanz der Lastverwaltung zu erhöhen.

Eine weitere Innovation dieses Lastverwaltungskonzepts ist ein Lastmodell für verteilte objektorientierte Systeme. Das Lastmodell liefert eine formale Darstellung aller Kenngrößen, die ein Lastbewertungsalgorithmus zur Entscheidungsfindung heranziehen kann. Das hier vorgestellte Lastmodell grenzt sich insofern von anderen Ansätzen ab, als es eine strikte Trennung der Lastbewertungsstrategie und ihrer Realisierung herbeiführt. Dadurch ist es möglich, Lastbewertungsstrategien zu entwickeln, ohne dabei konkrete Lastwerte betrachten zu müssen. Mit Hilfe des Lastmodells wurde ein Algorithmus zur Lastbewertung formuliert, der die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen unterstützt. Durch das formale und sehr detaillierte Lastmodell war es möglich, die Lastbewertungsstrategie mathematisch exakt darzustellen. Dies ist ein wesentlicher Vorteil gegenüber anderen Ansätzen, deren Lastbewertungsstrategien lediglich auf der Anwendung einzelner Heuristiken beruhen.

Das hier vorgestellte Lastverwaltungskonzept ist allgemein anwendbar und unterliegt keinerlei Einschränkungen hinsichtlich der unterstützten Anwendungen. Die Realisierung des Lastverwaltungskonzepts konzentriert sich auf den Bereich der wissenschaftlichen und unternehmensweiten Anwendungen, da dieser Bereich im Moment das Haupteinsatzgebiet für verteilte objektorientierte Systeme darstellt. Aus diesem Grund wurde CORBA (Common Object Request Broker Architecture) als Grundlage für eine Implementierung gewählt. Das hier entwickelte Lastverwaltungssystem trägt den Namen LMC (Load Managed CORBA). LMC setzt auf einer bestehenden

CORBA-Implementierung auf und erweitert diese um Funktionalität zur Lastverwaltung. Die Schnittstellen des Lastverwaltungssystems LMC fügen sich nahtlos in das Programmiermodell von CORBA ein, so daß sie problemlos in künftige Versionen des CORBA-Standards integriert werden könnten.

Eine Reihe von Untersuchungen in einer Testumgebung und eine Fallstudie aus dem Bereich der medizinischen Bildverarbeitung demonstrieren die praktische Anwendbarkeit und den Nutzen des hier entwickelten Lastverwaltungskonzepts. Das Lastverwaltungssystem LMC ist in der Lage, allen Formen der Lastungleichheit in verteilten objektorientierten Systemen wirkungsvoll entgegenzutreten:

- LMC reagiert auf Hintergrundlast, die von lokalen Anwendungen verursacht wird, indem es Objekte von überlasteten Rechnern auf weniger stark belastete Rechner migriert. Dies ermöglicht es, sowohl lokale als auch verteilte Anwendungen auf demselben Rechensystem zu betreiben.

- LMC berücksichtigt die spezifischen Ressourcenanforderungen der Anwendungen, indem es Objekte entsprechend ihrer Ressourcenanforderungen auf das Rechensystem verteilt. Damit können mehrere Anwendungen in demselben Rechensystem ausgeführt werden, um die Gesamtauslastung des Rechensystems zu erhöhen.

- LMC nimmt auch auf die Heterogenität des Rechensystems Rücksicht, so daß Rechner mit unterschiedlicher Leistungsfähigkeit genutzt werden können. Die Lastverwaltung verteilt die Anwendungen und ihre Objekte entsprechend ihres Ressourcenbedarfs auf die unterschiedlich leistungsfähigen Rechner.

- LMC gewährleistet, daß die Anfragelast einzelner Objekte auf mehrere Rechner verteilt wird. Dies verhindert, daß Objekte und ihre ausführenden Rechner aufgrund zu vieler Anfragen überlastet werden.

Zusammenfassend kann man sagen, daß LMC sämtliche Aspekte der Lastverwaltung in verteilten objektorientierten Systemen zur Laufzeit behandelt. Für den Entwickler ist dies transparent, d.h. die Problematik der Lastverteilung muß beim Software-Design nicht mehr berücksichtigt werden. Dies vereinfacht die Entwicklung verteilter objektorientierter Anwendungen drastisch.

Die durchweg sehr guten Ergebnisse, die mit dem Lastverwaltungssystem LMC erzielt wurden, sind einerseits auf seine effiziente Implementierung und andererseits auf das zugrundeliegende Lastverwaltungskonzept zurückzuführen. Insbesondere die kombinierte Anwendung unterschiedlicher Lastverteilungsmechanismen wie Initialplazierung, Migration und Replikation hat sich bewährt. Nur auf diese Weise ist es möglich, geeignet auf die unterschiedlichen Lastsituationen zu reagieren, die in verteilten objektorientierten Systemen auftreten. Darüber hinaus gewährleistet die Unterstützung zustandsloser und zustandsbehafteter Objekte, daß die Lastverwaltung das gesamte Spektrum verteilter objektorientierter Anwendungen bedienen kann. Wegen der Transparenz der Lastverwaltung kann sich der Entwickler vollständig auf das Design der Anwendungslogik konzentrieren, da alle Aspekte der Lastverteilung zur Lauf-

zeit behandelt werden. Insgesamt gesehen hat sich das hier präsentierte Lastverwaltungskonzept sehr gut bewährt und sollte deshalb in zukünftigen Arbeiten ausgebaut und erweitert werden.

8.2 Ausblick

Ein interessanter Ansatzpunkt für zukünftige Arbeiten ist eine Erweiterung des Lastmodells und seiner Realisierung:

- Bei der Realisierung des Lastmodells konnte lediglich die Prozessorlast betrachtet werden. Die Erfassung der Speicherlast auf Anwendungsebene ist sehr schwierig, da gängige Betriebssysteme prozeßorientiert arbeiten und damit lediglich Lastinformation über Server-Prozesse aber nicht über einzelne Objekte liefern. Ebenfalls problematisch ist die Messung der Netzwerklast auf der Ebene des Rechensystems. Gängige Betriebssysteme betrachten lediglich die zugeteilte Übertragungskapazität des Netzwerks. Zur Darstellung der Netzwerklast benötigt man jedoch die nachgefragte Kapazität. In zukünftigen Arbeiten könnten geeignete Mechanismen entwickelt werden, um die benötigte Lastinformation zu gewinnen. Dies erfordert keine Änderung der Lastbewertungsstrategie, da das Lastmodell die Strategie und ihre Realisierung strikt trennt.

- Ein weiterer Ansatzpunkt wäre eine Erweiterung der Lastbewertungsstrategie. Der hier verwendete Algorithmus zur Lastbewertung führt nur eine Lastverschiebeaktion pro Bewertungszyklus durch. Dadurch ist gewährleistet, daß die Lastbewertungsstrategie einfach und nachvollziehbar bleibt. Zukünftige Arbeiten könnten den Lastbewertungsalgorithmus dahingehend erweitern, daß er mehrere Aktionen pro Bewertungszyklus durchführen kann. Dadurch würde die Reaktionszeit der Lastbewertung erheblich verkürzt und die Qualität der Lastverwaltung verbessert.

Ein vielversprechendes Forschungsgebiet ist die Kombination der Lastverwaltung mit anderen Diensten und Protokollen:

- Wegewahl und Flußkontrolle in Netzwerken: Das hier vorgestellte Lastverwaltungskonzept verzichtet auf die Betrachtung der Netzwerktopologie, da globale Aspekte wie die Wegewahl und die Flußkontrolle in den Aufgabenbereich der Netzwerkprotokolle fallen. Dadurch kann die Lastverwaltung lediglich für eine gleichmäßige Auslastung einzelner Netzsegmente sorgen, indem sie Objekte zwischen überlasteten und unterlasteten Segmenten umverteilt. Die Steuerung der Datenströme zwischen den einzelnen Netzsegmenten ist jedoch die Aufgabe der Netzwerkprotokolle. Die kombinierte Betrachtung dieser Problemstellungen wäre für beide Seiten vorteilhaft.

- Ein-/Ausgabe: Die Ein-/Ausgabe in verteilten Systemen erfolgt häufig über Betriebssystem- oder Datenbankdienste. Diese Dienste sind jedoch nicht Be-

standteil einer verteilten Anwendung und unterliegen somit auch nicht der Kontrolle der Lastverwaltung. Vielmehr werden auch im Bereich der Datenbanksysteme große Anstrengungen zur Realisierung der Lastverteilung unternommen. Eine verbesserte Kooperation in beiden Teilgebieten würden die Wirksamkeit der Lastverwaltung erhöhen.

- Fehlertoleranz: Ein Mechanismus, der sowohl bei der Realisierung der Fehlertoleranz als auch bei der Lastverwaltung verwendet wird, ist die Replikation. Obwohl sich die Zielsetzungen und Anforderungen der Fehlertoleranz und der Lastverwaltung stark unterscheiden, bestehen Gemeinsamkeiten hinsichtlich der grundlegenden Mechanismen wie beispielsweise der Verwaltung von Replikatgruppen. Eine bessere Integration dieser Aufgabenstellungen würde die kombinierte Anwendung von Lastverwaltung und Fehlertoleranz ermöglichen.

Die Untersuchung dieser Problemstellungen und die Entwicklung geeigneter Lösungsansätze geht weit über den üblichen Rahmen der Lastverwaltung hinaus. Dies eröffnet jedoch vielversprechende Perspektiven, da von einer besseren Integration unterschiedlicher Dienste wesentliche Synergieeffekte zu erwarten sind. Letztendlich ist dies der einzige Weg, um den wachsenden Umfang und die steigende Komplexität der Anwendungen auch weiterhin beherrschbar zu halten.

Literaturverzeichnis

[1] C. AMZA, A. L. COX, S. DWARKADAS, P. KELEHER u. a. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer, 29(2)*, S. 18–28, 1996.

[2] H. E. BAL, R. BHOEDJANG, R. HOFMAN, C. JACOBS u. a. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems, 16(1)*, 1998.

[3] D. J. BARRETT, R. SILVERMAN u. M. LOUKIDES. *SSH, the Secure Shell – The Definitive Guide*. O'Reilly, 2001.

[4] T. BARTH, G. FLENDER, B. FREISLEBEN u. F. THILO. Load Distribution in a CORBA Environment. In *Proceedings of the 1st International Symposium on Distributed Objects and Applications (DOA'99)*. IEEE Press, 1999.

[5] BEA SYSTEMS INC. CORBA Component Model Joint Revised Submission. Technischer Bericht, OMG (Object Management Group), 1999. http://www.omg.org, orbos/99-07-01.

[6] BEA SYSTEMS INC. BEA MessageQ Release 5.0. Technischer Bericht, 2000. http://www.bea.com.

[7] A. BENSOUSSAN, C. T. CLINGEN u. R. C. DALEY. The Multics Virtual Memory. In *Proceedings of the 2nd ACM Symposium on Operating Systems Principles*. ACM, 1969.

[8] A. D. BIRELL u. B. J. NELSON. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems, 2(1)*, S. 39–59, 1984.

[9] K. P. BIRMAN u. R. VAN RENESSE. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Press, 1993.

[10] A. P. BLACK. Object Identity. In *3rd International Workshop on Object Orientation in Operating Systems*, S. 175–176. IEEE Press, 1993.

[11] S. H. BOKHARI. On the Mapping Problem. *IEEE Transactions on Computers, 30(3)*, S. 207–214, 1981.

[12] U. BORGHOFF u. J. SCHLICHTER. *Rechnergestützte Gruppenarbeit – Eine Einführung in Verteilte Anwendungen*. Springer Verlag, 1995.

[13] G. BROSE. JacORB: Implementation and Design of a Java ORB. In *International Conference on Distributed Applications and Interoperable Systems (DAIS'97)*. Chapman & Hal, 1997.

[14] F. BUSCHMANN, R. MEUNIER, H. ROHNERT, P. SOMMERLAD u. M. STAL. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons, 1996.

[15] T. L. CASAVANT u. J. G. KUHL. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering, 14(2)*, S. 141–154, 1988.

[16] K. CHANDY u. L. LAMPORT. Distributed Snapshots: Determing Global States of Distributed Systems. *ACM Transactions on Computer Systems, 3(1)*, S. 63–75, 1985.

[17] P. E. CHUNG, Y. HUANG, S. YAJNIK, D. LIANG u. a. DCOM and CORBA Side by Side, Step By Step, and Layer by Layer. *C++ Report*, Januar 1998.

[18] CISCO SYSTEMS. Load Balancing: A Solution for Improving Server Availability. Technischer Bericht, 2000. http://www.cisco.com.

[19] S. CLARKE. ObjectStore Data Management. Technischer Bericht, 2001. http://www.objectdesign.com.

[20] G. COULOURIS, J. DILLIMORE u. T. KINDBERG. *Distributed Systems: Concepts and Design*. Pearson Education Limited, Harlow, England, 1998.

[21] P. DASGUPTA, R. LEBLANC u. W. APPELBE. The Clouds Distributed Operating System: Functional Description, implementation details and Related Work. In *8th International Conference on Distributed Systems*. IEEE Press, 1988.

[22] A. DEARLE, R. DI BONA, J. FARROW, F. HENSKENS u. a. Grasshopper: An orthogonally persistent operating system. *Computing Systems, 7(3)*, S. 289–312, 1994.

[23] A. DICKMAN. Two-Tier Versus Three-Tier Applications. *Informationweek, 533*, S. 74–80, 1995.

[24] G. EDDON u. H. EDDON. *Inside Distributed COM*. Microsoft Press, 1998.

[25] D. L. EGAR, E. D. LAZOWSKA u. J. ZAHORJAN. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering, 12(5)*, S. 662–675, 1986.

[26] D. L. EGAR, E. D. LAZOWSKA u. J. ZAHORJAN. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, S. 63–72, Santa Fe, New Mexico, USA, 1988.

[27] J. EICHLER. Entwicklung und Evaluierung von Lastverteilungsstrategien für verteilte objektorientierte Systeme. Diplomarbeit, Technische Universität München, Deutschland, 2000.

[28] D. M. ETTER, D. C. KUNCICKY u. D. W. HULL. *Introduction to MATLAB 6*. Prentice Hall, 2001.

[29] T. EWALD. Use Application Center or COM and MTS for Load Balancing Your Component Servers. *Microsoft Systems Journal*, 2000. http://www.microsoft.com/.

[30] D. FERRARI u. S. ZHOU. An Empirical Investigation of Load Indices for Load Balancing Applications. In *12th International Symposium on Computer Performance Modeling, Measurement, and Evaluation*. Elsevier Science Publishers, 1988.

[31] G. C. FOX, R. D. WILLIAMS u. P. C. MESSINA. *Parallel Computing Works*. Morgan Kaufmann, San Francisco, USA, 1994.

[32] R. S. J. FRACKOWIAK, K. J. FRISTON, C. D. FRITH, R. J. DOLAN u. J. C. MAZZIOTTA. *Human Brain Function*. Academic Press, USA, 1997.

[33] K. J. FRISTON. Statistical Parametric Mapping and other Analysis of Functional Imaging Data. In A. W. TOGA u. J. C. MAZZIOTTA [Hrsg.], *Brain Mapping – The Methods*, S. 363–388. Academic Press, San Diego, USA, 1996.

[34] E. GAMMA, R. HELM, R. JOHNSON u. J. VLISSIDES. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[35] M. R. GAREY u. D. S. JOHNSON. *Computers and Interactability – A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, USA, 1979.

[36] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG u. a. [Hrsg.]. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[37] B. GOODHEART u. J. COX. *The Magic Garden Explained - The Internals of the UNIX System V Release 4*. Prentice Hall, 1994.

[38] A. GOSCINSKI. *Distributed Operating Systems*. Addison Wesley, 1991.

[39] A. GOSCINSKI. *Distributed Operating Systems – The Logical Design*. Addison Wesley, Sidney, Australien, 1991.

[40] J. GRAY u. A. REUTER. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, USA, 1993.

[41] R. GUERRAOUI u. A. SHIPER. Software-Based Replication for Fault Tolerance. In A. K. SOMANI u. N. H. VAIDYA [Hrsg.], *Special Edition on Fault Tolerance, IEEE Computer*. IEEE Press, 1997.

[42] S. HALABI u. D. MCPHERSON. *Internet Routing Architectures*. Cisco Press, 2000.

[43] M. HARCHOL-BALTER u. A. B. DOWNEY. Exploiting Lifetime Distributions for Dynamic Load Balancing. In *ACM Sigmetrics '96 Conference on Measurement and Modeling of Computer Systems*, S. 13–24, Philadelphia, USA, 1996. ACM.

[44] M. HEITING. Klassifikation und Evaluierung von Middleware für Client-Server-Umgebungen. Diplomarbeit, Technische Universität München, Deutschland, 1999.

[45] M. HENNING u. S. VINOSKI. *Advanced CORBA Programming with C++*. Addison Wesley, 1999.

[46] M. HENNING. Binding, Migration, and Scalability in CORBA. *Communications of the ACM, 41(10)*, 1998.

[47] J. L. HENNING. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, July 2000.

[48] Y. HIRANO, T. SATOH, U. INOUE u. K. TERANAKA. Load Balancing Algorithms for Parallel Database Processing on Shared Memory Multiprocessors. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, S. 210–217. IEEE Press, 1991.

[49] J. E. HOPCROFT u. J. D. ULLMAN. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Addison Wesley, 1993.

[50] C. S. HORSTMANN u. G. CORNELL. *Core Java 2*, Bd. 1: Fundamentals. Sun Microsystems Press, 1999.

[51] C. S. HORSTMANN u. G. CORNELL. *Core Java 2*, Bd. 2: Advanced Features. Sun Microsystems Press, 1999.

[52] C. HUITEMA u. A. FRANCE. *Routing in the Internet*. Prentice Hall, 1995.

[53] INTERNATIONAL BUSINESS MACHINES CORPORATION. MQSeries Version 5.2 Release Guide. Technischer Bericht, 2000. http://www.ibm.com.

[54] IONA TECHNOLOGIES. OrbixNames Programmer's and Administrator's Guide. Technischer Bericht, 2000. http://www.iona.com.

[55] IONA TECHNOLOGIES. Orbix 2000 Programmer's Guide. Technischer Bericht, 2001. http://www.iona.com.

[56] L. J. KEEDY u. J. ROSENBERG. Support for Objects in the Monads Architecture. In *Proceedings of the 3rd Workshop on Persistent Object Systems*. Springer Verlag, 1989.

[57] W. KENT. A Rigorous Model of Object References, Identity, and Existence. *Object-Oriented Programming 4(3)*, 1991. http://home.earthlink.net/~billkent/Doc/obrefidx.htm.

[58] W. KENT. A Behavioral View of Object State. Technischer Bericht, Hewlett-Packard Laboratories, 1992. http://home.earthlink.net/~billkent/Doc/obstate.htm.

[59] W. KENT. The State of Object Technology. *Canadian Information Processing*, 1992. http://home.earthlink.net/~billkent/Doc/obstate.htm.

[60] S. KHOSHAFIAN u. G. COPELAND. Object Identity. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*. ACM, 1986.

[61] W. G. KREBS. Queue Load Balancing / Distributed Batch Processing and Local RSH Replacement System. Technischer Bericht, 1998. http://www.gnuqueue.org/home.html.

[62] O. KREMIEN u. J. KRAMER. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems, 3(6)*, S. 747–760, 1992.

[63] S. J. LEFFLER. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison Wesley, 1989.

[64] R. LELAND u. B. HENDRICKSON. An Empirical Study of Static Load Balancing Algorithms. In *Proceedings of the Scalable High-Performance Computing Conference*, S. 682–685. IEEE Press, 1994.

[65] W.E. LELAND u. T.J. OTT. Load-balancing Heuristics and Process Behavior. In *Proceedings of the Performance '86 and ACM Sigmetrics '86, Joint Conference on Computer Performance Modeling, Measurement and Evaluation*, S. 54–69, Raleigh, North Carolina, USA, 1986. Association for Computer Machinery, New York, USA.

[66] M. LINDERMEIER. Untersuchungen zur werkzeuggestützten Abschätzung des Leistungsverhaltens paralleler Programme unter veränderten Last- und Abarbeitungsbedingungen. Diplomarbeit, Technische Universität München, Deutschland, 1997.

[67] M. LINDERMEIER. Load Management for Distributed Object-Oriented Environments. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA'2000)*, S. 59–68, Antwerpen, Belgien, 2000. IEEE Press.

[68] M. J. LITZKOW u. M. LIVNY. Experience with the Condor Distributed Batch System. In *IEEE Workshop on Experimental Distributed Systems*, Huntsville, Alabama, USA, 1990. IEEE Press.

[69] M. LOOMIS. *Object Databases – The Essentials*. Addison Wesley, 1996.

[70] C. LOOSLEY u. F. DOUGLAS [Hrsg.]. *High-Performance Client/Server*. John Wiley & Sons, 1998.

[71] T. LUDWIG. *Automatische Lastverwaltung für Parallelrechner*. Doktorarbeit, Technische Universität München. BI-Wissenschaftsverlag, Deutschland, 1993.

[72] T. LUDWIG, M. LINDERMEIER, A. STAMATAKIS u. G. RACKL. Tool Environments in CORBA-based Medical High Performance Computing. In *Proceedings of the 6th International Conference on Parallel Computing Technologies (PaCT-2001)*, Novosibirsk, Russia, 2001. Springer Verlag.

[73] U. MAIER. *Konzepte zur Ressourcenverwaltung für wissenschaftlich-technische Anwendungen in verteilten Rechensystemen*. Doktorarbeit, Technische Universität München. Shaker Verlag, Aachen, Deutschland, 1999.

[74] C. MARCHETTI, M. MECELLA, A. VIRGILLITO u. R. BALDONI. An Interoperable Replication Logic for CORBA Systems. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA'2000)*, Antwerpen, Belgien, 2000. IEEE Press.

[75] D. R. MAURO u. K. J. SCHMIDT. *Essential SNMP*. O'Reilly, 2001.

[76] M. MAY. Vergleich von PVM und CORBA bei der verteilten Berechnung medizinischer Bilddaten. Diplomarbeit, Technische Universität München, Deutschland, 2000.

[77] L. MERZ u. H. JASCHEK. *Grundkurs der Regelungstechnik – Einführung in die praktischen und theoretischen Methoden*. Oldenbourg Verlag, 1996.

[78] MICROSOFT CORPORATION. DCOM Architecture. Technischer Bericht, 1998. http://www.microsoft.com.

[79] MICROSOFT CORPORATION. Microsoft Application Center 200 Component Load Balancing Technology Overview. Technischer Bericht, 2000. http://www.microsoft.com/.

[80] L. E. MOSER, P. M. MELLIAR-SMITH u. V. AGRAWALA. Processor Membership in Asynchronous Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems, 5(5)*, S. 459–473, 1994.

[81] S. J. MULLENDER [Hrsg.]. *Distributed Systems*. Addison Wesley, 2. Aufl., 1993.

[82] F. MUNZ. *Parallele und verteilte Algorithmen in der funktionellen medizinischen Bildgebung*. Doktorarbeit, Technische Universität München. Shaker Verlag, Aachen, Deutschland, 2000.

[83] B. NATARAJAN, A. GOKHALE, D. C. SCHMIDT, M. KIRCHER u. J. PARSONS. DOORS: Towards High-performance Fault-Tolerant CORBA. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA'2000)*, Antwerpen, Belgien, 2000. IEEE Press.

[84] B. NICHOLS, D. BUTTLAR u. J. P. FARRELL. *Pthreads Programming*. O'Reilly, 1996.

[85] D. A. NICHOLS. Using Idle Workstations in a Shared Computing Environment. *Operating Systems Review*, S. 5–12, 1987.

[86] OBJECTIVITY INC. Objectivity Technical Overview – Release 6.0. Technischer Bericht, 2001. http://www.objectivity.com.

[87] B. OESTEREICH. *Objektorientierte Softwareentwicklung - Design und Analyse mit der Unified Modeling Language*. Oldenbourg Verlag, 1998.

[88] OMG (OBJECT MANAGEMENT GROUP). Common Facilities Architecture. Technischer Bericht, 1995. http://www.omg.org.

[89] OMG (OBJECT MANAGEMENT GROUP). A Discussion of the Object Management Architecture. Technischer Bericht, 1997. http://www.omg.org.

[90] OMG (OBJECT MANAGEMENT GROUP). CORBA Services: Common Object Services Specification. Technischer Bericht, 1998. http://www.omg.org.

[91] OMG (OBJECT MANAGEMENT GROUP). Objects by Value – Joint Revised Submission. Technischer Bericht, 1998. http://www.omg.org.

[92] OMG (OBJECT MANAGEMENT GROUP). Fault Tolerant CORBA Specification. Technischer Bericht, 1999. http://www.omg.org, orbos/99-12-08.

[93] OMG (OBJECT MANAGEMENT GROUP). Persistent State Service 2.0 Specification. Technischer Bericht, 1999. http://www.omg.org, orbos/99-07-07.

[94] OMG (OBJECT MANAGEMENT GROUP). RFI: Supporting Aggregated Computing in CORBA. Technischer Bericht, 1999. http://www.omg.org, orbos/99-01-04.

[95] OMG (OBJECT MANAGEMENT GROUP). Externalization Service 1.0 Specification. Technischer Bericht, 2000. http://www.omg.org, formal/2000-06-16.

[96] OMG (OBJECT MANAGEMENT GROUP). Life Cycle Service 1.0 Specification. Technischer Bericht, 2000. http://www.omg.org, formal/2000-06-18.

[97] OMG (OBJECT MANAGEMENT GROUP). Persistent Object Service 1.0. Technischer Bericht, 2000. http://www.omg.org, formal/2000-06-21.

[98] OMG (OBJECT MANAGEMENT GROUP). The Common Object Request Broker: Architecture and Specification – Revision 2.5. Technischer Bericht, 2001. http://www.omg.org.

[99] A. ORAM [Hrsg.]. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[100] O. OTHMAN, C. O'RYAN u. D. C. SCHMIDT. An Efficient Adaptive Load Balancing Service for CORBA. *Distributed Systems Engineering Journal*, 2001.

[101] O. OTHMAN, C. O'RYAN u. D. C. SCHMIDT. The Design of an Adaptive CORBA Load Balancing Service. *Distributed Systems Engineering Journal*, 2001.

[102] S. PETRI. *Lastausgleich und Fehlertoleranz in Workstation-Clustern*. Shaker Verlag, Aachen, Deutschland, 1997.

[103] F. PLASIL u. M. STAL. An Architectural Overview of Distributed Objects and Components in CORBA, Java RMI, and COM/DCOM. *Software Concepts & Tools, 19(1)*, 1998.

[104] PYTHON SOFTWARE FOUNDATION. Python Library Reference – Release 2.2.1. Technischer Bericht, 2001. http://www.python.org.

[105] G. RACKL. *Monitoring and Managing Heterogeneous Middleware*. Doktorarbeit, Technische Universität München. Shaker Verlag, Aachen, Deutschland, 2001.

[106] G. RACKL, M. LINDERMEIER, M. RUDORFER u. M. SÜSS. MIMO - An Infrastructure for Monitoring and Managing Distributed Middleware Environments. In *Middleware 2000 - IFIP/ACM International Conference on Distributed Systems Platforms*, Bd. 1795, Lecture Notes in Computer Science, S. 71–87. Springer Verlag, 2000.

[107] G. RACKL, T. LUDWIG, M. LINDERMEIER u. A. STAMATAKIS. Efficiently Building On-Line Tools for Distributed Heterogeneous Environments. In *International Workshop on Performance-Oriented Application Development for Distributed Architectures, Perspectives for Commercial and Scientific Environments (PADDA 2001)*, Technische Universität München, 2001.

[108] E. RAHM. Dynamic Load Balancing in Parallel Database Systems. In *Proceedings of Euro-Par'96 Parallel Processing*, Bd. 1123, Lecture Notes in Computer Science. Springer Verlag, 1996.

[109] C. RÖDER. *Load Management Techniques in Distributed Heterogeneous Systems*. Doktorarbeit, Technische Universität München. Shaker Verlag, Aachen, Deutschland, 1998.

[110] C. RÖDER, T. LUDWIG u. A. BODE. Flexible Status Measurement in Heterogeneous Systems. In H. R. ARABNIA [Hrsg.], *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Bd. I, S. 247–254, Las Vegas, Nevada, USA, 1998. CSREA Press.

[111] P. RECHENBERG u. G. POMBERGER. *Informatik-Handbuch*. Hanser Verlag, München, Deutschland, 1997.

[112] R. VAN RENESSE, K. P. BIRMAN u. S. MAFFEIS. Horus, a Flexible Group Communication System. *Communications of the ACM, 39(4)*, 1996.

[113] W. ROSENBERRY, D. KENNEY u. G. FISHER. *Understanding DCE*. O'Reilly, 1992.

[114] B. SCHIEMANN. *Intelligente Lastverteilung für Datenbank-Management-Systeme*. Doktorarbeit, Technische Universität München, Deutschland, 1997.

[115] D. C. SCHMIDT, D. L. LEVINE u. S. MUNGEE. The Design and Performance of Real Time Object Request Broker. *Computer Communications, 21(4)*, S. 294–324, 1998.

[116] A. SCHMIDT. Konzepte zur Realisierung von Objektpersistenz in verteilten objektorientierten Umgebungen. Diplomarbeit, Technische Universität München, Deutschland, 2000.

[117] F. B. SCHNEIDER. Implementing Fault-Tolerant Services Using the State-Machine Approach: A Tutorial. *ACM Computing Surveys*, 1990.

[118] T. SCHNEKENBURGER u. G. STELLNER. *Dynamic Load Distribution for Parallel Applications*. Teubner-Texte, Leipzig, Deutschland, 1997.

[119] T. SCHNEKENBURGER. The ALDY Load Distribution System. Technischer Bericht, SFB 342/11/95 A, Technische Universität München, Deutschland, 1995.

[120] B. SCHÄTZ. *Ein methodischer Übergang von asynchron zu synchron kommunizierenden Systemen*. Doktorarbeit, Technische Universität München, 1998. http://www4.informatik.tu-muenchen.de/papers/Schaetz98.html.

[121] B. SHIRAZI, A. HURSON u. K. KAVI. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Press, 1995.

[122] V. SINGHAL, S. KAKKAD u. P. WILSON. Texas: An Efficient, Portable Persistent Store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, S. 11–33, San Miniato, Italien, 1992.

[123] M. SNIR, S. OTTO, S. HUSS-LEDERMAN, D. WALKER u. J. DONGARRA [Hrsg.]. *MPI: The Complete Reference*. MIT Press, 1996.

[124] P. SOUSA, A. RITO u. J. MARQUES. Object Identifiers and Identity: A Naming Issue. In *4th International Workshop on Object Orientation in Operating Systems*, S. 127–129. IEEE Press, 1995.

[125] W. STALLINGS. *Operating Systems – Internals and Design Principles*. Prentice Hall, 1998.

[126] W. STALLINGS. *Snmp, Snmpv2, Snmpv3, and Rmon 1 and 2*. Addison Wesley, 1998.

[127] A. STAMATAKIS. Interoperable Tool Deployment for the Late Development Phases of Distributed Object-Oriented Programs. Diplomarbeit, Technische Universität München, Deutschland, 2001.

[128] G. STELLNER u. J. PRUYNE. Resource Management and Checkpointing for PVM. In *Proceedings of the Second European PVM User Group Meeting*, S. 131–136, Lyon, Frankreich, 1995. Editions Hermes.

[129] G. STELLNER. CoCheck: Checkpointing and Process Migration for MPI. In *International Parallel Processing Symposium*, S. 526–531, Los Alamitos, USA, 1996. IEEE Press.

[130] W. R. STEVENS. *UNIX Network Programming*, Bd. 1: Networking APIs - Sockets and XT1. Prentice Hall, 2. Aufl., 1997.

[131] W. R. STEVENS. *UNIX Network Programming*, Bd. 2: Interprocess Communications. Prentice Hall, 2. Aufl., 1998.

[132] SUN MICROSYSTEMS. RPC: Remote Procedure Call Specification Version 2. IETF RFC 1831. Technischer Bericht, 1995. http://www.ietf.org/rfc/rfc1831.txt.

[133] SUN MICROSYSTEMS. XDR: External Data Representation Standard. IETF RFC 1832. Technischer Bericht, 1995. http://www.ietf.org/rfc/rfc1832.txt.

[134] SUN MICROSYSTEMS. Java Native Interface Specification – Revision 1.1. Technischer Bericht, 1997.

[135] SUN MICROSYSTEMS. Java Object Serialization – Revision 1.4. Technischer Bericht, 1997.

[136] SUN MICROSYSTEMS. Java Remote Method Invocation Specification – Revision 1.2. Technischer Bericht, 1997.

[137] A. S. TANENBAUM. *Moderne Betriebssysteme*. Hanser Verlag, München, Deutschland, 1994.

[138] M. M. THEIMER u. B. HAYES. Heterogeneous Process Migration by Recompilation. In *11th International Conference on Distributed Computing Systems*, S. 18–27. IEEE Press, 1991.

[139] G. D. VAN ALBADA, J. CLINCKEMAILLIE, A. H. L. EMMEN, J. GEHRING u. a. Dynamite - blasting obstacles to parallel cluster computing. In M. BOASSON, J. A. KAANDORP, J. F. M. TONINO u. M. G. VOSSELMAN [Hrsg.], *Proceedings of the 5th Annual Conference of the Advanced School for Computing and Imaging (ASCI)*, S. 31–37, Delft, Niederlande, 1999. ASCI.

[140] F. VAUGHAN u. A. DEARLE. Supporting Large Persistent Stores Using Conventional Hardware. In *5th International Workshop on Persistent Object Systems*. Springer Verlag, 1992.

[141] VERSANT INC. Versant Reference Manual. Technischer Bericht, 2001. http://www.versant.com.

[142] VERSANT INC. Versant Usage Manual. Technischer Bericht, 2001. http://www.versant.com.

[143] VISIGENIC SOFTWARE INC. Distributed Object Computing in the Internet Age. Technischer Bericht, 1997. http://www.borland.com.

[144] K. WALDSCHMIDT [Hrsg.]. *Parallelrechner – Architekturen - Systeme - Werkzeuge*. Teubner Verlag, Stuttgart, Deutschland, 1995.

[145] R. P. WEICKER. An Overview of Common Benchmarks. *IEEE Computer, 23(12)*, S. 65–76, 1990.

[146] WELLCOME DEPARTMENT OF COGNITIVE NEUROLOGY, UNIVERSITY COLLEGE LONDON. Statistical Parametric Mapping. Technischer Bericht, 2001. http://www.fil.ion.ucl.ac.uk/spm.

[147] X/OPEN LDT. X/Open Common Application Environment – Distributed Transaction Processing: Reference Model. Technischer Bericht, Berkshire, England, 1991.

Index

I

J

K

L

M

O

P

R

S

T

V

W

Z