Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Modulare Mechanismen zur Lastbalancierung in Peer-to-Peer Systemen

Verfasser:

Karl-Heinz Deutinger

30. Juli 2005

Betreuer:

Dipl.-Inf. Sven Apel

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Deutinger, Karl-Heinz:
Modulare Mechanismen zur Lastbalancierung in Peer-to-Peer Systemen
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2005.

Vorwort

Die vorliegende Diplomarbeit ist im Rahmen des Studiengangs Wirtschaftsinformatik an der Universität Magdeburg entstanden. Die Ausgabe des Themas erfolgte am 01. Februar 2005. Im Rahmen eines Laborpraktikums hat der Autor in einer kleinen Gruppe eine Groupwareapplikation auf Basis des an der Universität Magdeburg entwickelten Peer-to-Peer Systems implementiert. In der vorliegenden Arbeit wird nun der Kern dieses Systems erweitert.

Danksagung

Für die intensive und sachkundige Betreuung dieser Diplomarbeit bedanke ich mich herzlich bei Dipl.-Inf. Sven Apel. Seine Anregungen und konstruktive Kritik über die gesamte Bearbeitungszeit hin haben mir sehr geholfen, diese Arbeit zu erstellen. Diese außerordentlich gute Betreuung halte ich keineswegs für selbstverständlich. Außerdem bedanke ich mich bei meiner Freundin Katharina, die mir rund um mein Diplom jede mögliche Unterstützung zukommen ließ. Meiner Mutter danke ich dafür, dass sie mir so viel private Arbeit wie möglich abgenommen hat, damit ich ausreichend Zeit hatte, mein Diplom rechtzeitig zu beenden. Meinem Vater danke ich für intensives Korrekturlesen. Sein geschultes Auge findet auch in Texten, die ich für fehlerfrei halte, noch ausreichend Raum für Verbesserungen. Zuletzt möchte ich noch meinen Freunden und Arbeitskollegen für dringend benötigtes Mitleid danken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Abkürzungen

AHEAD	Algebraic Hierarchical Equations for Application Design
AOP	Aspektorientierte Programmierung
ATS	AHEAD Tool Suite
CAN	Content Adressable Network
CPU	Central Processing Unit
CVS	Concurrent Versions System
DARPA	The Defense Advanced Research Projects Agency
DHT	Distributed Hash Table
ITI	Institut für Technische und Betriebliche Informationssysteme
IP	Internet Protocol
GCC	GNU Compiler Collection
FOP	Featureorientierte Programmierung
JNI	Java Native Interface
JVM	Java Virtual Machine
ON	Overlay Network
P2P	Peer-to-Peer
P2P-DS	Peer-to-Peer Datenstruktur
SETI	Search for Extraterrestrial Intelligence
TCP	Transmission Control Protocol
W3C	The World Wide Web Consortium

Kapitel 1

Einleitung

1.1 Motivation und Zielsetzung

Peer-to-Peer-Systeme (P2P-Systeme), auch synonym P2P-Netze genannt, stehen derzeit stark im Interesse der wissenschaftlichen Forschung. Im Gegensatz zu herkömmlicher *Client-Server* Struktur ist jeder Teilnehmer (Peer) eines P2P-Systems gleichzeitig Anbieter und Konsument von Dienstleistungen. Die häufigste Anwendung findet sich momentan in verschiedenen Dateitauschbörsen. Viele P2P-Netze zeichnen sich durch positive Eigenschaften wie zum Beispiel Skalierbarkeit, Fehlertoleranz oder technische und ökonomische Effizienz aus. Diese Vorteile finden sich besonders in strukturierten P2P-Systemen, die auch als P2P-Systeme der nächsten Generation *(Next-Generation P2P-Systems)* bezeichnet werden. Sie eröffnen neue Einsatzgebiete für P2P-Systeme, wie zum Beispiel das *Semantic Web* oder *Internet Computing*. Im Gegensatz zu unstrukturierten P2P-Systemen werden bei *Next-Generation P2P-Systems* verteilte Hashtabellen (DHT, engl.: *distributed hash table*) zur Speicherung und Auffindung von Daten verwendet, wobei jeder Peer einen genau definierten Datenbereich dieser Hashtabelle verwaltet. Der Zugriff auf die Daten findet über eindeutig zugeordnete Schlüssel statt. Eine Hashfunktion ermittelt die korrekten Schlüssel. Das Netz selber kümmert sich darum, bei welchem Peer es Daten abfragt oder einfügt. Dies führt zu erheblichen Vorteilen bei der Kommunikation der einzelnen Peers untereinander: Bei vielen Operationen im Netz (zum Beispiel der Suche nach Daten) findet ein intelligentes *Routing* der Anfragen statt. Dabei wird die Schlüsselraumgeometrie ausgenutzt, um Anfragen auf direktem Wege weiterzuleiten. In unstrukturierten P2P-Systemen werden im Allgemeinen alle bekannten Peers mit den Anfragen geflutet, was zwar etwas schneller ist, aber schnell zu erheblichen Einschränkungen führt. So leidet die Skalierbarkeit deutlich, da jeder Peer (fast) jede Anfrage bearbeiten muss und dadurch mit steigender Netzgröße schnell an die Grenzen seiner Leistungsfähigkeit stößt.

An der Universität Magdeburg wurde ein *Content-Adressable-Network* (CAN) [RFH+01] in einer flexiblen, hoch konfigurierbaren Architektur implementiert. Dazu existieren proprietäre Erweiterungen wie zum Beispiel Kontakt-*Caching* für effizientes *Rou-*

ting [BB03], Reputationsmechnismen [BB04a] oder Metadatenverwaltung [BA05]. Als Vertreter der strukturierten P2P-Netze kann es deren Vorteile einer großen Bandbreite verteilter Anwendungen zur Verfügung stellen. Die vorliegende Arbeit beschäftigt sich mit der Verbesserung der Architektur und des Protokolls dieses CAN mit Hilfe von Lastbalancierung.

Prinzipiell kann sich die Leistungsfähigkeit eines P2P-Netzes durch Lastbalancierung erhöhen, wenn Teilnehmer des Systems an ihrer Leistungsgrenze arbeiten und Anfragen nur verzögert oder gar nicht mehr bearbeiten. Eine Verbesserung kann dadurch erreicht werden, dass weniger belastete Peers den höher belasteten einen Teil der Belastung (belegte Ressourcen allgemein, z.B. Massenspeicher oder Rechenleistung) abnehmen und damit eine möglichst gleichmäßige Auslastung aller Peers erreicht wird. Da die gesamte Leistungsfähigkeit eines P2P-Netzes mit der Anzahl seiner Peers steigt, skaliert ein lastbalanciertes Netzwerk sehr gut, auch wenn wenige, bestimmte Daten besonders gefragt sind.

Wenn bei *Filesharing*-Diensten die Nachfrage nach einigen wenigen Dateien besonders hoch ist, beispielsweise weil diese sehr beliebt sind, so bricht die Datenübertragungsrate zu jedem Peer aufgrund der begrenzten Leitungskapazitäten im allgemeinen stark ein. Ein Replikationsmechanismus kann dafür sorgen, dass diese Dateien schnellstmöglich an wenige andere Teilnehmer verteilt werden, so dass sich die große Nachfrage durch eine Verteilung auf mehrere Peers schneller befriedigen lässt.

Im Rahmen dieser Arbeit werden Definitionen im Bereich Lastbalancierung sowie verschiedene Möglichkeiten, Last zu bestimmen („Lastarten“), erarbeitet. Eine Analyse der Lastarten verdeutlicht, welche von ihnen gemessen werden müssen, um bestimmte Lastbalancierungsziele zu erreichen. Anschließend werden verschiedene Lastbalancierungsmethoden betrachtet. Einige der Möglichkeiten zur Lastbestimmung und der Methoden zur Lastbalancierung werden exemplarisch implementiert. Unter Berücksichtigung von realen Anwendungsszenarien wird deren Wirksamkeit mittels aussagekräftiger Experimente überprüft.

Die Lastbalancierung gehört jedoch nicht zu den Kernfunktionen eines CAN, es handelt sich vielmehr um eine Spezialfunktion (engl.: *special purpose concern*). Eine saubere Trennung der Spezialfunktionen bei der Implementierung propagiert das Prinzip des *separation of concerns* [Dij76]. Es sagt aus, dass durch bestmögliche Aufteilung von Quellcode nach Programmfunktionen großen Anwendungen zu hoher Qualität und guter Wartbarkeit verholfen werden kann. Allerdings müssen mit der im Java-Umfeld üblichen objektorientierten Vorgehensweise viele Basisklassen erweitert und verändert werden, was diesem Prinzip widerspricht. Diese Schwierigkeiten haben auch Mitarbeiter der Universität Magdeburg bei der Implementierung eines CAN erkannt. Es wurde daher auf eine Architektur umgestellt, die die Trennung des Quellcodes nach Funktionalität ermöglicht. Dazu wurde die sogenannte featureorientierte Programmierung (FOP) eingesetzt. Die Flexibilität der bestehenden Architektur soll nicht eingeschränkt werden. Die Implementierung eines oder mehrerer Lastbalancierungsalgorithmen bringt in jedem Fall zusätzliche Kommunikation zwischen den Peers mit sich, da zum Beispiel jeder Peer

möglichst aktuelle Informationen über die Belastung anderer Peers benötigt. In Umgebungen mit langsamen Verbindungen zwischen den Peers kann Lastbalancierung also komplett unerwünscht sein.

Um die Vorteile der Trennung der Funktionen im Quellcode zu nutzen, erfolgt die Implementierung der Lastbalancierung durch aspektorientierte Programmierung (AOP). Im Gegensatz zur objektorientierten Programmierung kapseln dabei sogenannte Aspekte Funktionen eines Systems, wo sonst eine Erweiterung vieler verstreuter, existierender Klassen notwendig wäre. Eine Aktivierung, Deaktivierung, Erweiterung oder Verbesserung bestimmter Funktionen ist so an einer Stelle im Quellcode möglich. Allerdings wurde bisher wenig über Nachteile oder Erfolge bei der Erweiterung großer, bereits realisierter Projekte außerhalb der Implementierung der Protokollierung geschrieben (Ausnahme in [CC04]).

Diese Arbeit verfolgt zwei wesentliche Ziele in unterschiedlichen Bereichen:

- **Softwaretechnik**

  Ein Ziel besteht in der Bewertung der Eigenschaften aspektorientierter Programmierung in einem realen und nicht-trivialen Projekt. Die bisherigen Erfahrungen der Arbeitsgruppe mit dem CAN versprechen ein positives Ergebnis. Es wird aber darüber hinaus untersucht, welche Schwierigkeiten auftreten, und wie effektiv AOP im Bereich strukturierter P2P-Systeme einsetzbar ist. Zudem wird der Einsatz von AOP im Zusammenhang mit FOP bewertet.

- **Lastbalancierung**

  Das zweite Ziel besteht in der Untersuchung verschiedener Lastbalancierungsalgorithmen. Dazu gehört die Definition von Grundbegriffen im Bereich Lastbalancierung sowie eine Analyse der verschiedenen Ziele, die mit ihr verfolgt werden können. Außerdem werden unterschiedliche Möglichkeiten, Last zu messen, untersucht. Um die Effektivität der Lastbalancierung zu evaluieren, müssen praxisrelevante Experimente entwickelt und durchgeführt werden. Die Implementierung verschiedener Methoden zur Lastmessung und -balancierung erfolgt jedoch nur prototypisch. Das bedeutet, dass nur eine geringe Anzahl dieser Methoden implementiert wird. Die Architektur und der Experimentaufbau sollen eine Erweiterung ohne großen Aufwand ermöglichen.

1.2 Überblick

Kapitel 2. Hier werden zunächst die für die folgende Arbeit notwendigen Grundlagen vermittelt. Zum einen werden verschiedene Arten von P2P-Systemen und ihre Basisfunktionen vorgestellt. Außerdem werden die wünschenswerten Eigenschaften definiert, sowie der Zusammenhang zur Lastbalancierung hergestellt. Zum anderen soll ein Exkurs in die Forschung an biologischen Algorithmen aufzeigen, welche Verbesserungen durch sie in P2P-Systemen erreicht werden können. Abschließend werden softwaretechnische

Grundlagen zu Programmfamilien sowie featureorientierter und aspektorientierter Programmierung erklärt, die dem Prinzip des *separation of concerns* genügen.

Kapitel 3. Dieses Kapitel bietet eine Einführung in den Aufbau des an der Universität Magdeburg implementierten CAN. Insbesondere werden die Struktur und die für die Lastbalancierungsimplementierung relevanten Teile des Systems vorgestellt.

Kapitel 4. Klare Definitionen von Grundbegriffen im Bereich Lastbalancierung werden in Kapitel 4 erarbeitet. Einer Analyse und Systematisierung verschiedener Lastarten und existierender Methoden zur Lastbalancierung folgt abschließend die Auswahl der zu implementierenden Algorithmen.

Kapitel 5. Danach werden Details zum Entwurf und der Implementierung der Lastbalancierung besprochen. Dazu wird eine flexibel erweiterbare und konfigurierbare Architektur entwickelt. Außerdem wird die Implementierung von Lastdatenverteilung und Lastmessungen sowie der eigentlichen Methoden zur Lastbalancierung besprochen. Dabei wird jeweils parallel die konkrete Nutzung von featureorientierter und aspektorientierter Programmierung vorgestellt.

Kapitel 6. Die Wirksamkeit der implementierten Methoden zur Lastbalancierung wird in diesem Kapitel anhand verschiedener Experimente nachgewiesen. Dafür werden zunächst „ideale“ Anwendungsszenarien entworfen, die voraussichtlich sehr gut auf Lastbalancierung reagieren. Danach werden dann Experimente mit realistischen Daten durchgeführt. Der Interpretation der Ergebnisse dieser Experimente folgt eine Evaluierung der verwendeten softwaretechnischen Methoden.

Kapitel 7. Abschließend werden in Kapitel 7 die Ergebnisse der Arbeit zusammenfasst. Dazu wird ein Ausblick gegeben, an welcher Stelle zukünftige Arbeiten ansetzen können, um weitere Verbesserungen im Bereich Lastbalancierung in strukturierten P2P-Systeme zu erreichen.

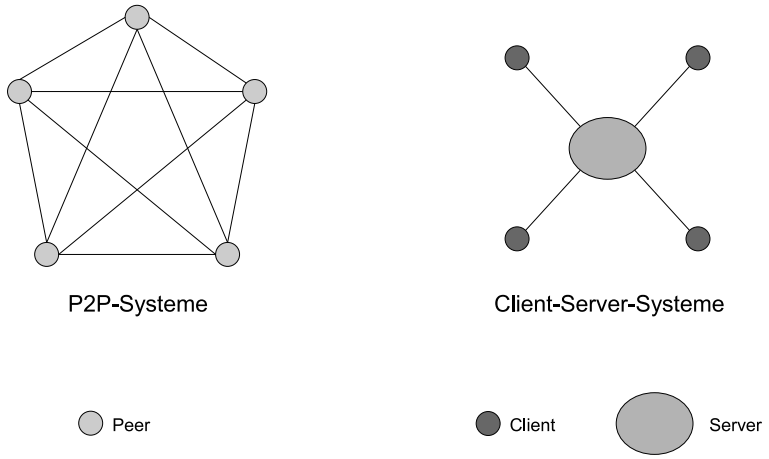

Abbildung 2.1: Struktur von P2P- und Client-Server-Systemen

Kapitel 2

Grundlagen

2.1 Peer-to-Peer-Systeme

Ein Peer-to-Peer-System ist „ein System mit vollständig dezentraler Selbstorganisation und Ressourcenverwaltung“ ([SW04]), bei dem jeder Teilnehmer sowohl Dienste anbietet als auch nutzt. Im Gegensatz dazu sind diese Rollen bei Client-Server-Netzen für jeden Teilnehmer eindeutig festgelegt (vgl. Abbildung 2.1). Beispielanwendungen, die auf P2P-Netzen aufbauen, beschreibt Tabelle 2.1.

In [SW04] werden wichtige Vorteile von P2P-Netzen beschrieben, zu denen unter anderem Fehlertoleranz, Skalierbarkeit, Ausfallsicherheit und Gerechtigkeit (hier im Sinne von: „Alle arbeiten, alle profitieren“) gehören. Diese Eigenschaften werden durch Vermeidung einer zentralen Verwaltung oder eines zentralen Diensteanbieters, durch Anreizmechanismen und verschiedene Protokolle erreicht. Der Ausfall eines einzelnen Peers führt somit nicht zum Ausfall des kompletten Netzes, sondern nur eines kleinen Teils der Funktionalität beziehungsweise der Daten. Insbesondere im Hinblick auf gezielte, verteil-

Anwendungsbereich	Beispielapplikation(en)
Distributed Storage	Oceanstore, Gnutella, Napster
Kollaborationssysteme	Endeavors Magi Enterprise, Groove Virtual Office
Telefonnetze/-anlagen	Nimcat Networks nimX
Ortungsbasierte Systeme	Fraunhofer Institut Standpersonalfinder, DARPA selbstheilende Minenfelder
Web-ähnliche Systeme (*Distributed Content*)	Neurogrid, W3C Amaya

Tabelle 2.1: Beispiele für P2P-Applikationen

te Angriffe (*Distributed-denial-of-service*-Attacken) gegen zentrale Dienste kann dadurch höhere Ausfallsicherheit gewährleistet werden. Die Fehlertoleranz kann durch Replikationsmechanismen erhöht werden. Skalierbarkeit wird erreicht, da jeder Peer nur einen kleinen Teil des Netzes beziehungsweise der anfallenden Anfragen und Daten kennt. Bei einer sehr hohen Anzahl von Peers wird der dadurch ebenfalls sehr hohe Koordinationsaufwand so aufgeteilt, dass jeder einzelne Peer nicht mehr Arbeit übernehmen muss als in kleinen Netzen. Dadurch, dass jeder Teilnehmer nur Leistungen aus dem Netz entnehmen kann, wenn er sie gleichzeitig auch bereitstellt, erreicht man höhere Gerechtigkeit gegenüber herkömmlichen Systemen. Allerdings „[treffen] oftmals nur im Idealfall alle Eigenschaften gleichzeitig [zu]“ ([SW04]).

[SHS05] beschreibt, dass es keine exakte Definition des Begriffs P2P gibt, da es sich um ein Paradigma handelt, das auf verschiedenen Ebenen zum Einsatz kommen kann. Tabelle 2.2 erläutert, um welche technischen Ebenen es sich dabei handelt.

Die grundsätzliche Aufgabe eines P2P-Systems besteht in der gemeinsamen Nutzung beziehungsweise Verknüpfung folgender Ressourcen (vgl. [DGC+04]):

- **Verteilte Rechenleistung**

  Algorithmen, die sich in möglichst viele, unabhängige Teile zerlegen lassen, können stark von der Rechenleistung der großen Zahl an Peers in einem P2P-Netz profitieren. Beispiele hierfür finden sich in der Klimaforschung, dem Rendering vom 3D-Grafiken oder dem SETI-Projekt[1].

- **Verteilte Speicherkapazität**

  Die bekanntesten Filesharing-Dienste setzen auf eine P2P-Architektur, die auf verteilte Speicherkapazität ausgerichtet ist. Dabei tauschen die einzelnen Peers Daten mit anderen Peers. Zur Erhöhung der Austauschgeschwindigkeit kann parallel auch auf mehrere Peers zugegriffen werden, die im Besitz der gewünschten Datei(en) sind.

[1] `http://www.seti.org`

Ebene	Erklärung/Beispielapplikation
Benutzerebene	Ein Groupwaresystem kann trotz Client/Server-Architektur auf Benutzerebene als P2P-Systeme beschrieben werden. Alle Teilnehmer können auf die gleiche Weise agieren, also beispielsweise Nachrichten empfangen/versenden oder Termine koordinieren. Beispiele sind Kombinationen aus Microsoft Outlook und Microsoft Exchange.
Applikationsebene	Hierzu zählen die klassischen Filesharing-Dienste wie Napster oder eDonkey. Diese Systeme stehen in dem Kontext, in dem in der Öffentlichkeit von P2P-Systemen gesprochen wird. Auch in dieser Arbeit werden P2P-Netze auf dieser Ebene besprochen.
Datenverwaltungsebene	Ein Anwendung auf Basis eines CAN verteilt die Datenverwaltung auf mehrere Computer, die gleichberechtigt jeweils Teilbereiche der Daten verwalten. Daher wird hier von P2P auf Datenverwaltungsebene gesprochen werden.

Tabelle 2.2: Ebenen des P2P-Paradigmas

2.1.1 Strukturierte und nicht-strukturierte P2P-Netze

- Bei **unstrukturierten** P2P-Systemen verwaltet kein Peer genau definierte Datenbereiche und weiß deshalb auch nicht, welche Daten ein anderer anbieten kann. Da deswegen zum Beispiel bei einer Suche jeder Peer abgefragt werden muss, sind diese Netze sehr schlecht skalierbar, da sie die Netzwerkverbindungen schnell überlasten. Viele Systeme arbeiten mit sogenannten Superpeers, die die Daten mehrerer anderer Peers sammeln, um diesem Nachteil entgegen zu wirken. Damit bauen sie allerdings wieder eine Art Client-Server-Subnetz auf und untergraben so einige der Vorteile von P2P-Systemen.

- Peers in **strukturierten** P2P-Systemen hingegen sind logisch durch ein virtuelles Netzwerk *(Overlay Network*, ON*)* oberhalb der eigentlichen Netzwerkinfrastruktur – üblicherweise einem IP-basiertes Netzwerk – verbunden. Nachbarschaft in einem *Overlay Network* muss dabei nicht im Zusammenhang mit Nachbarschaft im physischen Netz stehen, da ein *Overlay Network* verschiedene Möglichkeiten hat, seine Peers anzuordnen. Beispiele für Topologien sind stern-, baum- oder ringförmige Netzwerke. Das CAN, das im Rahmen dieser Arbeit erweitert wird, ordnet seine Peers in einem n-dimensionalen Torus an. Eine genaue Beschreibung verschiedener Topologien sowie deren Vor- und Nachteile bieten Milojicic et al. in [MKL+05]. In allen strukturierten P2P-Systemen verwalten die einzelnen Peers

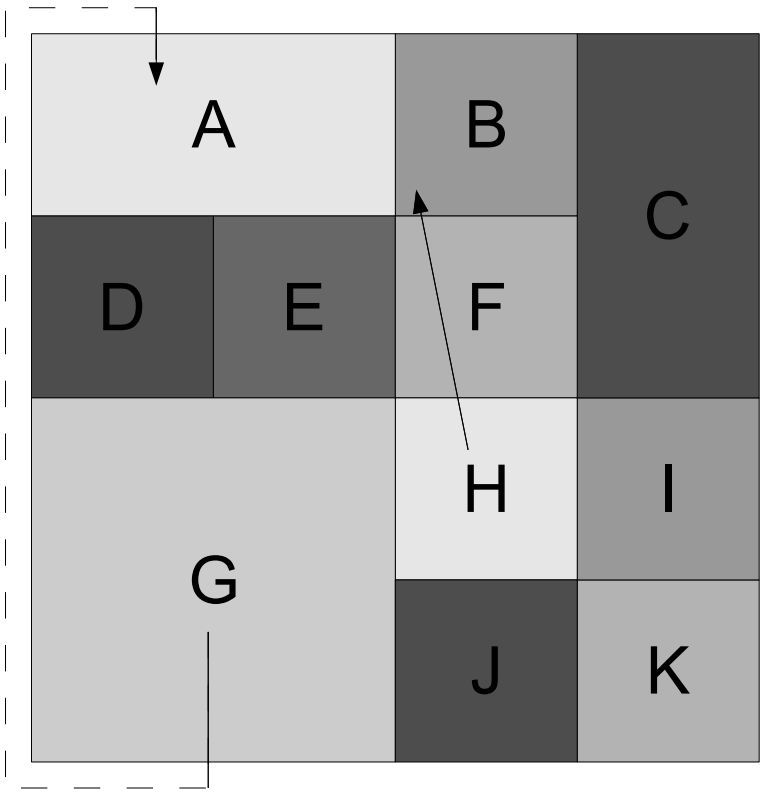

Abbildung 2.2: Zonenaufteilung und Routing in einem 2-dimensionalem CAN (vgl. [ATS04])

einen eindeutig definierten Datenbereich und kennen nur ihre Nachbarn und deren Datenbereiche. Man spricht bei diesen strukturierten P2P-Systemen auch von P2P-Datenstrukturen, da sich der gesamte Datenbereich der Peers ähnlich wie eine Hashtabelle ansprechen lässt. Bekommt ein Peer eine Anfrage bezüglich eines bestimmten Schlüssels, der außerhalb seines eigenen Bereichs liegt, leitet er sie an an einen anderen Peer weiter. Der genaue Weiterleitungsalgorithmus kann unterschiedlich implementiert sein. Beispiele wären Algorithmen, die sich an der Distanz zwischen Zielschlüssel und den Schlüsselbereichen der Nachbarn orientieren oder den Zielpeer mit einer Hashfunktion ermitteln (vgl. [AB05]). Die Vorteile von strukturierten P2P-Systemen beschreiben Steinmetz und Wehrle in [SW04]. So sind sie unstrukturierten P2P-Systemen vor allem im Bereich Skalierbarkeit, aber auch in Leistungsfähigkeit und Fehlertoleranz überlegen. Allerdings gibt es auch noch offene Probleme, die in [DGMY03] erläutert werden. Dazu gehört vor allem die Problematik der Sicherheit (vgl. [SM02], [CDG+02]). So kann zum Beispiel ein böswilliger Peer das ganze System mit Anfragen überlasten oder falsche Daten zurückliefern. Ansätze zur Bekämpfung böswilliger Peers werden unter anderem in [BB04a] beschrieben.

- In dieser Arbeit wird an einem **CAN**, einer möglichen Topologie eines strukturier-

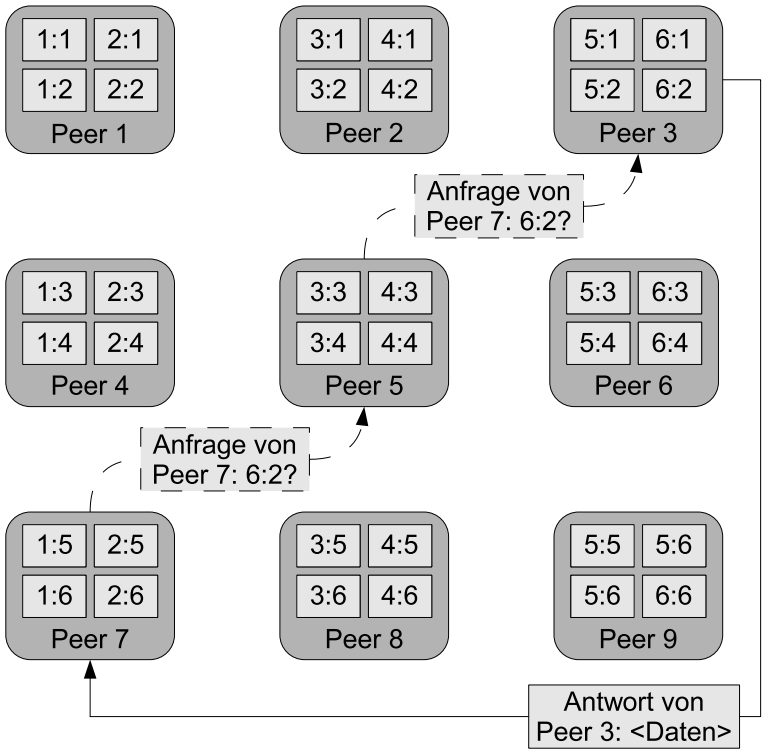

Abbildung 2.3: Anfrage und Antwort in einem 2-dimensionalem CAN

ten P2P-Systems, gearbeitet. Ein CAN ordnet seine Peers in einem n-dimensionalen Torus. Dabei wird die Geometrie ausgenutzt, um beim Routing die kürzesten Wege zu finden. In Abbildung 2.2 wird dieses Prinzip am Beispiel eines zweidimensionalen CAN illustriert. Eine Anfrage geht von Zone H aus. Diese Anfrage wird dabei über Zone F an den Zielbereich in Zone B weitergeleitet. Durch die Torus-Form kann die Anfrage aus der Zone G direkt an den Zielbereich in A geleitet werden.

Die Datenverteilung auf verschiedene Peers sowie den Weg einer Nachricht zeigt Abbildung 2.3. Dort fragt ein Peer (Peer 7) nach den Daten, die im Schlüssel 6:2 abgelegt sind. Diese Anfrage wird in einer Nachricht über Peer 5 an den Zielpeer 3, der den entsprechenden Schlüsselbereich verwaltet, weitergeleitet. Die Antwort mit den Daten des Schlüssels 6:2 schickt Peer 3 dann direkt an Peer 7.

2.1.2 Grundlegende Funktionen eines P2P-Systems

[CDJD03] beschreibt die folgenden grundlegenden Funktionen, die ein P2P-System bieten kann:

- **Routing**

  Das Routing ist dafür zuständig, einen „ausreichend guten“ Weg zu finden. Kriterien für gute Pfade können zum Beispiel geringe Verzögerung oder geringe Anzahl

an Knotenpunkten, die durchlaufen werden müssen, sein.

- **Suche**

  Eine erfolgreiche Suche in einem P2P-System liefert einem Peer die Ressourcen, nach denen er gesucht hat, beziehungsweise die Orte, an denen sie zu finden sind.

- **Monitoring**

  Als Monitoring wird die Überwachung der „Gesundheitszustände“ eines P2P-Systems bezeichnet. Dazu gehört sowohl das Messen eines oder verschiedener Indikatoren als auch das Interpretieren der Messwerte. Beide Funktionen sind aufgrund der Charakteristika von P2P-Systemen nicht-trivial, da der Zusammenhang zwischen einzelnen Peers und dem gesamten Netzwerk hergestellt werden muss.

- **kollektive Berechnungen**

  Funktionen, deren Ergebnis auf dem Status der Peers basieren, und die auf verteilte Weise berechnet werden, werden als kollektive Berechnungen bezeichnet. Beispiele sind Berechnungen der kürzest möglichen Pfade nach dem Link-State-Algorithmus [Tan03], die Berechnung der Durchschnittslast [JMB03] oder die oben genannte Funktion der Interpretation von Gesundheitsindikatoren.

- **Topologie Management**

  Die Anordnung der Peers in einem virtuellen Netzwerk ist nicht vorgegeben, sondern muss erstellt und verwaltet werden. Daher gehört das Topologie Management ebenfalls zu den Funktionen eines P2P-Systems.

2.1.3 Wünschenswerte Eigenschaften von P2P-Systemen

[CDC+04] beschreibt die folgenden wünschenswerten Eigenschaften von P2P-Systemen:

- **Selbstorganisation**

  Ein System wird als selbstorganisierend beschrieben, wenn im Startzustand jeder Peer ausschließlich über lokale Informationen (wie eine Liste mit Nachbarn) verfügt und im Laufe der Entwicklung des Systems alle Operationen lokal erfolgen (also über die eigenen Nachbarn). Ein solches System ist in der Lage, sich selbst auf Basis von lokalen Regeln zu reorganisieren. Für sich genommen ist nicht sofort zu erkennen, warum es sich dabei um eine wünschenswerte Eigenschaft handelt. Allerdings lassen sich andere wünschenswerte Eigenschaften nur sehr schwer oder gar nicht ohne Selbstorganisation realisieren.

- **Anpassungsfähigkeit**

  Anpassungsfähige Systeme sind in der Lage, auf Änderungen in der Umgebung so zu reagieren, dass es nach einer bestimmten Zeitspanne in etwa dieselbe Leistungsfähigkeit zeigt. Dabei spielen zwei Aspekte eine Rolle: Zum einen die Zeit, die ein

System braucht, um nach einer Umgebungsänderung wieder die gleiche Leistungsfähigkeit zu besitzen. Zum anderen die Empfindlichkeit, mit der ein System auf eine Änderung in der Umgebung reagiert, also die Stärke des Leistungseinbruchs.

- **Selbstheilung**

  Ohne Selbstheilung wäre eine P2P-Datenstruktur dauerhaft beschädigt, wenn ein Peer ausfällt. Sein Datenbereich wäre unverwaltet, dies würde zu fehlerhafter Funktion des Systems führen. Selbstheilungsmechanismen können Schäden durch den Ausfall einzelner Peers durch Replikation verhindern oder Folgefehler durch Neustrukturierung vermeiden.

- **hohe Skalierbarkeit**

  Ein hoch skalierbares P2P-System verringert seine Leistungsfähigkeit (fast) nicht bei zunehmender Größe. Alternativ kann synonym zu Skalierbarkeit auch der Begriff Wachstumspotenzial verwendet werden. Normalerweise wird die Größe von P2P-Systemen durch die Anzahl ihrer Peers beschrieben, möglich wären aber auch andere Variablen wie zum Beispiel die Menge der verwalteten Daten. Ein im Bereich Performance gut skalierendes System gilt aber nicht als allgemein gut skalierend, wenn mit steigender Netzgröße der Ressourcenverbrauch extrem stark ansteigt. Unstrukturierte P2P-Systeme gelten deshalb als schlecht skalierend, da sie Anfragen an (fast) alle anderen Peers weiterleiten und damit die Kommunikationsleitungen schnell verstopfen.

Eine Balancierung oder Verteilung von Last verbessert ein System in allen diesen wünschenswerten Eigenschaften. Erst die Lastbalancierung macht ein P2P-System **anpassungsfähig** (nach der obigen Definition), da auf andere Weise nur sehr schwer auf Veränderungen der Nachfrage reagiert werden kann (eine Erweiterung der Hardware eines Hochlastpeers ist Verschwendung, wenn andere Peers noch ausreichend Reserven besitzen). Obwohl es kein Kernpunkt von Lastbalancierung ist, können in diesem Rahmen entwickelte Replikationsmechanismen auch zur **Selbstheilung** eingesetzt werden, wenn einzelne Peers ausfallen. Ein Datenverlust ist dann nahezu ausgeschlossen. Auch hohe **Skalierbarkeit** kann nur sinnvoll mit Hilfe von Lastbalancierung erreicht werden. Ohne sie werden Peers, die besonders stark nachgefragte Daten verwalten, durch steigende Netzgröße immer stärker belastet und können die Anfragen irgendwann gar nicht mehr oder nur noch sehr langsam beantworten. Selbstverständlich müssen diese Verbesserungen durch Implementierung von Lastbalancierungsmethoden **selbstorganisierend** geschehen.

Zur vollen Ausnutzung der verteilten Ressourcen ist die Implementierung von Lastbalancierungsalgorithmen essentiell. Sind einige Peers in einem auf **verteilte Rechenleistung** ausgerichteten P2P-Netz stark überlastet, wird die gesamte Berechnung viel später beendet als bei gleichmäßiger Auslastung. In P2P-Systemen mit **verteilter Speicherkapazität** kann die Suche beziehungsweise der Transport von besonders gefragten Daten schneller stattfinden.

Einen tieferen Einblick in Peer-to-Peer-Systeme liefern unter anderem [SHS05], [SFT02], [VS05] oder [Ora01].

2.2 Exkurs: Biologische Algorithmen in P2P-Systemen

Ein aktuelles Thema der Forschung ist die Adaption von biologischen Vorgehensweisen für IT-Systeme. [CDEM03] und [CDG+04] stellen folgende fünf Beispiele für biologische Algorithmen, die für P2P-Systeme interessant sind, vor:

- **Ameisen bei der Suche nach Futter**

  Wenn eine Ameise auf Futtersuche geht, hinterlässt sie duftende Pheromone. Nachfolgende Ameisen wählen mit höherer Wahrscheinlichkeit stärker duftende Wege. Existieren mehrere Futterquellen, wird der kürzeste Weg durch höhere Frequentierung in derselben Zeit stärker markiert. So gehen schnell fast alle Ameisen auf diesem kürzesten Weg. Auf vergleichbare Weise kann ein lastbalanciertes Routing in P2P-Systemen implementiert werden. Sind Peers so stark ausgelastet, dass sie Nachrichten nur verzögert weiterleiten, so werden diese Peers durch den Ameisenalgorithmus oftmals ausgelassen und somit entlastet. Es wurde bereits von Di Caro und Dorigo in [CD98b] und [CD98a] gezeigt, dass der Ameisenalgorithmus das bestehende Internet-Routing verbessern kann.

- **Immunzellen bei der Suche nach und Abwehr von Antigenen**

  Immunzellen suchen nach Antigenen mit einer bestimmten Struktur, die sie durch passende Antikörper neutralisieren können. Außerdem werden durch Mutation und Selektion ständig bessere Immunzellen entwickelt und produziert, die wesentlich schneller auf dieselben Antigene reagieren. In einem P2P-System kann die Immunreaktion für die Suche verwendet werden. Die gesuchten Objekte sind dann die Antigene, während das Suchmuster dazu passende Antikörper darstellt. Entsprechend dem lernfähigen Immunsystem könnte sich die Topologie eines P2P-Systems anhand bereits bearbeiteter Suchanfragen verändern.

- **Amöben bei der Suche nach neuen Futterquellen**

  In Umgebungen mit wenig Nahrung gehen Amöben zu kollektivem Verhalten über. Sie senden dabei ein Streusignal aus, das andere Amöben, die es empfangen, weiterleiten. Dadurch wird eine kollektive Berechnung des durchschnittlichen Sättigungsgrades ausgeführt. Durch Zusammenschluss versuchen sie nun, schnell neue Futterquellen zu finden. Im Bereich der Lastbalancierung könnte diese Methode bei der Berechnung der durchschnittlichen Last Anwendung finden.

- **Neuronale Netzwerke bei der Informationsübermittlung**

  In neuronalen Netzwerken kommunizieren immobile Neuronen über ein Netzwerk. Aus dem Status der einzelnen Neuronen bildet sich ein kollektives Gedächtnis. In

P2P-Systemen können auf diese Weise unter anderem Lastinformationen gespeichert werden.

- **Epidemische Ausbreitung von Viren**

  Vereinfacht ausgedrückt haben Viren kein Gedächtnis, sind aber mobil, können sich vermehren und sterben. Mit diesen simplen Techniken können sie große Systeme effizient infizieren. Dadurch können zum Beispiel Lastinformationen in P2P-Systemen verbreitet oder Ausfälle erkannt werden.

In dieser Arbeit wird allerdings weniger auf die biologischen Algorithmen eingegangen, da der Fokus auf der effektiven Implementierung bestehender Lastbalancierungsalgorithmen liegt. Obwohl einige der vorgestellten Ideen bei der Implementierung umgesetzt werden, soll dieser Abschnitt in erster Linie das Potenzial von biologischen Algorithmen für das Thema Lastbalancierung aufzeigen. Damit soll für zukünftige Arbeiten in dieser Richtung eine Grundlage gebildet werden.

2.3 Softwaretechnik

Dieser Abschnitt beschreibt die softwaretechnischen Grundlagen, die benötigt werden, um die Implementierung der Lastbalancierung in dem an der Universität Magdeburg entwickeltem P2P-System nachvollziehen zu können. Da es sich bei dem P2P-System um eine Programmfamilie handelt wird zunächst dieser Themenbereich erläutert. Diese Programmfamilie verwendet in hohem Maße featureorientierte und aspektorientierte Programierungstechniken. Daher werden auch hier allgemeine Grundlagen sowie konkrete Werkzeuge, die bei der Implementierung benutzt wurden, vorgestellt.

2.3.1 Programmfamilien und das *separation of concerns*-Prinzip

Bei Programm- oder Produktfamilien handelt es sich um verschiedene Softwareprogramme, die auf einer gemeinsamen Infrastruktur basieren. Dieses Konzept wurde erstmals von Parnas in [Par76] präsentiert und ist vergleichbar mit Produktlinien aus der Güterproduktion. Auch dort wird eine gemeinsame Basis benutzt, auf der viele spezielle Anforderungen realisiert werden. Man verspricht sich davon schnellere und kostengünstigere Entwicklung bei hoher Qualität. Programmfamilien bauen auf einer Hierarchie von wiederverwendbaren Funktionalitäten auf. Einzelne Funktionalitäten werden dabei möglichst getrennt von anderem Quellcode implementiert. Dadurch wird gleichzeitig das Prinzip des *separation of concerns* ([Dij74]) umgesetzt, das Vorteile im Bezug auf Wartbarkeit und Qualität auch bei großen Projekten verspricht. Trotz seines Alters[2] ist dieses Thema nicht zuletzt aufgrund steigender Komplexität[3] hochaktuell. Ein Beispiel für die

[2] Dijkstra entwickelte es bereits 1974.

[3] Der wertmäßige Anteil von Elektronik in Automobilen beispielsweise lag schon 1999 bei etwa 25%. In Zukunft soll er noch deutlich steigen ([PK99]).

erfolgreiche Umsetzung von Programmfamilien zeigen Liu und Batory in [LB04]. Eine Softwaresuite mit fünf Programmen spart durch eine gemeinsame Basis knapp 70% an Quellcode ein. Gleichzeitig konnte die zum Kompilieren benötigte Zeit um knapp 40% reduziert werden. Auch das an der Universität Magdeburg entwickelte CAN basiert auf einer Programmfamilie. Mittels einfacher Änderungen kann zum Beispiel die gleiche Basis verwendet werden, um P2P-Systeme mit anderen Topologien zu erstellen oder um speziellen Anwendungswünschen wie zum Beispiel Sicherheit oder *Caching* zu genügen. Im Rahmen dieser Arbeit wird diese Programmfamilie dahin gehend erweitert, dass sie bei Bedarf Lastbalancierung anbietet, um den Anforderungen von stark unter Last stehenden P2P-Systemen gerecht zu werden.

2.3.2 Featureorientierte Programmierung

Unter featureorientierter Programmierung (FOP) versteht man die wohlgeplante und -definierte Aufteilung von Softwarebestandteilen nach deren Funktion (engl.: *Feature*).[4] Dadurch lässt sich das Prinzip des *separation of concerns* umsetzen.

Ein bekanntes Designmodell für die featureorientierte Programmierung ist GenVoca, das Batory und O'Malley in [BO92] entwickelt haben. GenVoca ähnelt dem klassischen *step-wise refinement* (in etwa: schrittweise Verfeinerung), das Dijkstra in [Dij76] beschreibt. Allerdings sollen die Erweiterungen nicht in sehr vielen kleinen Stufen stattfinden, sondern immer zusammengefasst in einem sogenannten *Layer* (dt.: Ebene). In einer möglichen Implementierungsvariante wird eine neue Funktion dabei durch einen *Mixin Layer* definiert. Dieser kapselt die für diese neue Funktion notwendigen Klassenfragmente *(Mixins)*. Zum Zeitpunkt der Konfiguration wird durch die ausgewählte Hierarchie der *Mixin Layer* die jeweilige Elternklasse der einzelnen *Mixins* bestimmt. Bei der Programmgenerierung kann mittels einfacher Konfigurationsanweisungen bestimmt werden, welche Funktionen das fertige Programm enthalten soll.

Abbildung 2.4 verdeutlicht das *Mixin Layer*-Prinzip. Die Basis eines Programms beziehungsweise einer Programmfamilie besteht aus den Klassen A, B und C, die im *Layer* X zusammengefasst sind. *Feature* Y erweitert die Klasse A und C, *Feature* Z die Klassen B und C.

Mit einer einfachen Konfigurationsänderung bei der Programmerstellung kann darüber entschieden werden, ob das Programm die Funktionen X, Y, beide oder keine enthält. Eine detailliertere Einführung in diese Thematik bieten unter anderem [BO92] oder [BSR03].

Die P2P-Programmfamilie ist auf Basis der *AHEAD Toolkit Suite* (ATS) implementiert, die das *Mixin Layer*-Prinzip umsetzt. Dort arbeitet man mit .jak-Dateien, die im wesentlichen aus reinem Java-Quellcode bestehen, aber um zwei Schlüsselwörter erweitert wurden:

[4] Das Wort Funktion beziehungsweise *feature* wird in dieser Arbeit im Sinne einer Anforderung eines Benutzers an ein Programm, also auf intuitive Weise aus Sicht des Benutzers verwendet.

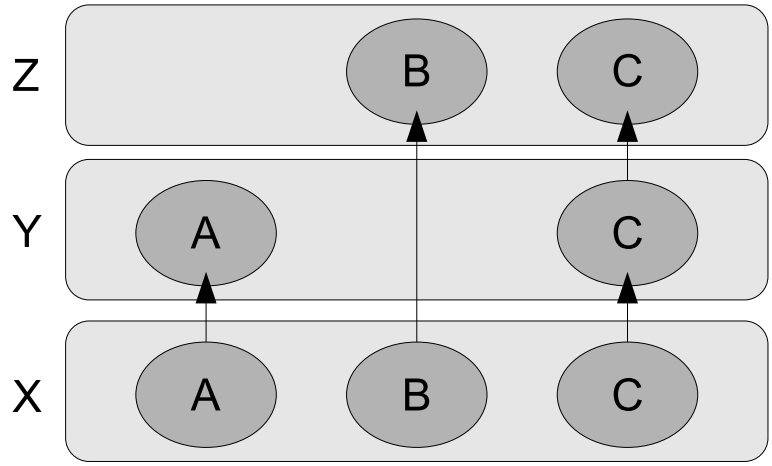

Abbildung 2.4: Mixin Layer-Beispiel

- **layer**

  Die *layer*-Anweisung definiert den aktuellen Quellcode als zu einer bestimmten Funktionalität zugehörig. Eine Lastbalancierungsimplementierung würde im einfachsten Fall mit dem Layer *loadbalancing* implementiert. Das bedeutet, dass jede Quellcodedatei, die für diese Implementierung gebraucht wird, mit der Zeile

  ```
  layer loadbalancing;
  ```

  beginnt, um sie entsprechend zu kennzeichnen. Diese Information wird dann später vom ATS-Codegenerator ausgewertet, um die Lastbalancierung je nach Konfiguration zum Programm hinzuzufügen oder sie zu ignorieren.

- **refines**

  Mit *refines* werden bereits existierende Klassen gekennzeichnet, die in diesem *Mixin* aber erweitert beziehungsweise neu definiert werden. Das Konzept ähnelt der Vererbung, allerdings wird stets derselbe Klassenname verwendet. Er enthält jedoch je nach Konfiguration mehr, weniger oder anders implementierte Funktionalität. Eine Erweiterung der Klasse *Peer* definiert man mit ATS mit folgendem Code in der Datei Peer.jak:

  ```
  refines class Peer {
    ...
    //Erweiterungen für eine bestimmte Funktion
    ...
  }
  ```

Durch den Codegenerator *(composer)* wird auf Basis einer Konfigurationsdatei automatisch der gewünschte Java-Quellcode synthetisiert. Dieser kann dann mit dem Java-Compiler *(javac)* übersetzt werden. Die Konfigurationsdatei beschreibt hierarchisch anhand der Namen der einzelnen Features das Zielprogramm. Im allgemeinen benutzt man für jedes *Feature* ein eigenes Unterverzeichnis. Darunter kann man auch Teilfunktionen auf die gleiche Weise strukturieren. Jede Klasse, die in einem *Layer* definiert oder erweitert wird, ist dort durch genau eine Datei mit dem Namen <Name der Klasse>.jak definiert. Abbildung 2.5 zeigt, wie die Datei- und Verzeichnisstruktur im Beispiel von Abbildung 2.4 aussehen würde:

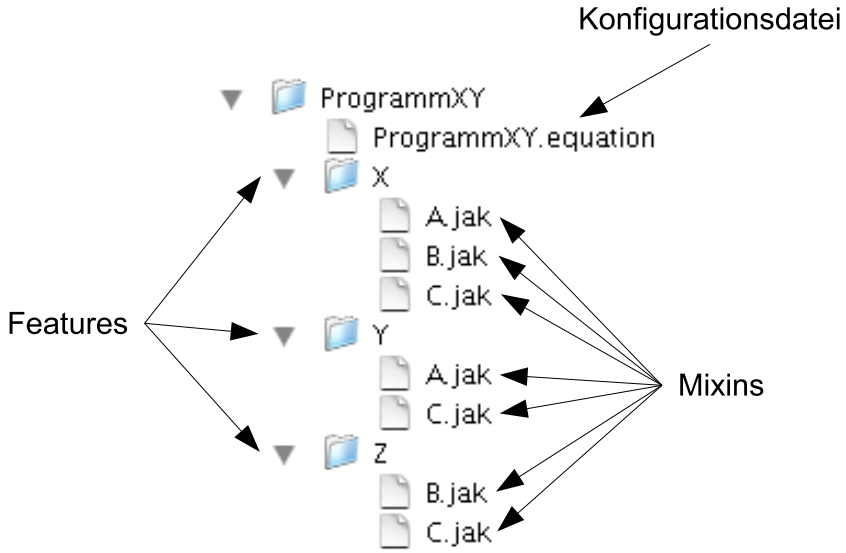

Abbildung 2.5: Beispiel-Verzeichnisstruktur bei FOP

Die Konfigurationsdatei *ProgrammXY.equation* bestimmt, welche *Features* in welcher Reihenfolge in den Programmen verfügbar sein sollen. In der ATS besteht sie aus einer einfachen Folge der Namen der *Features* (die im allgemeinen den Verzeichnisnamen entsprechen). Sollen alle *Features* benutzt werden, sieht sie wie folgt aus:

```
X Y Z
```

2.3.3 Aspektorientierte Programmierung

Eine weitere Technik zur Separierung der implementierten Programmfunktionen ist die aspektorientierte Programmierung. Während FOP im Rahmen einer gut geplanten Architektur mit aufeinander aufbauenden Funktionen recht universell eingesetzt werden kann, liegt der Schwerpunkt bei AOP auf „nicht-vorhersehbaren“ *Features* und Querschnittsfunktionen[5]. Neben dem in Abschnitt 1.1 genannten Beispiel der Protokollierung

[5]Unter Querschnittsfunktionen versteht man Funktionen eines Systems, die viele oder alle Funktionen des Systems betreffen beziehungsweise erweitern.

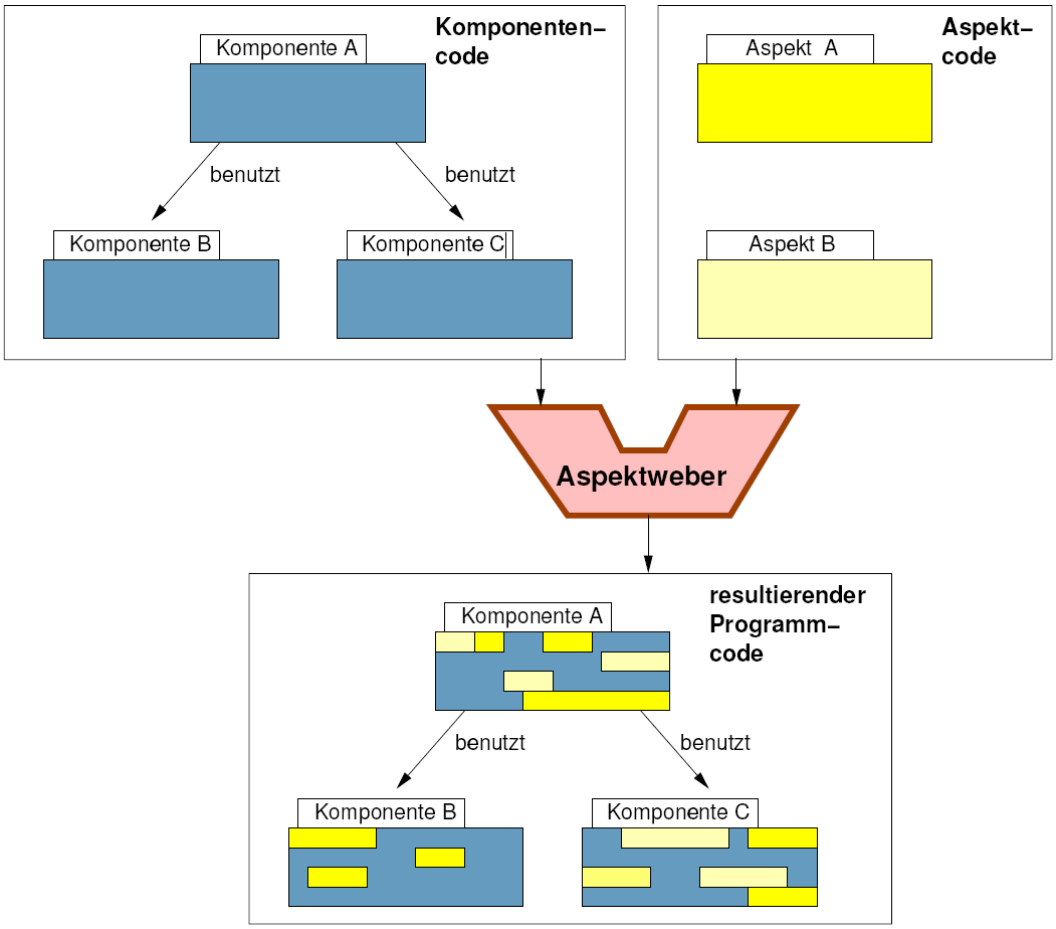

Abbildung 2.6: Der Vorgang des Aspektwebens (aus [Spi02])

gehören dazu auch Fehlerbehandlung, Debugging-Unterstützung, Änderungsverfolgung und Performance-Optimierung [6]. Normalerweise verteilt sich Quellcode, der sich mit diesen Themen beschäftigt, über viele Teile des Systems. Aspektorientierte Programmierung erlaubt es, diese Funktionen zu kapseln. So kann AOP, wie auch FOP, dem *separation of concerns*-Prinzip genügen. Dazu wird das Konzept der *joinpoints* eingeführt. Ein *joinpoint* ist ein bestimmter Teil eines Programms, der von besonderem Interesse ist. Ein oder mehrere (verstreute) *joinpoints* können von einem Aspekt aufgegriffen werden, um diese Programmteile zu erweitern oder zu ersetzen. Dabei ist ein Aspekt ähnlich wie eine Klasse aufgebaut. Ein „Weber“ webt den Aspektcode zu verschiedenen Zeitpunkten (beispielsweise vor der Kompilierung oder zur Laufzeit) in die Klassen, in die er eingreift, ein. Abbildung 2.6 veranschaulicht dieses Prinzip.

AspectJ implementiert AOP für Java. Es erweitert Java im wesentlichen um *pointcuts* und *advices*. Dabei definieren *pointcuts* bestimmte *joinpoints*. Dazu gehören vor allem

[6] Details und Beispielimplementierungen finden sich unter anderem auf `http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/starting.html`

Methodenaufrufe, aber unter anderem auch die Behandlung von *exceptions* oder Zugriffe auf Variablen. Bei der Definition der gewünschten Punkte können auch Platzhalter verwendet werden. Ein Beispiel für einen *pointcut* wäre der Aufruf jeder beliebigen Methode der Klasse Test:

```
pointcut alles(): call(* Test.*(..))
```

Mit *call* wird das Ziel – jeder Methodenaufruf der Klasse `Test` – bestimmt. In diesem Fall werden Methoden mit beliebiger Rückgabe (erster Stern), beliebigem Namen (zweiter Stern) und beliebigen Argumenten (die zwei Punkte) berücksichtigt. Das folgende Beispiel dagegen greift ausschließlich, wenn die Methode *testx* der Klasse TestX mit genau einer Zahl als Argument aufgerufen wird und eine Zahl zurückgibt:

```
pointcut x(): call(int TestX.testx(int))
```

Advices sind Anweisungen, die bestimmen, wann genau *pointcuts* ausgeführt werden. Dazu gehören *before* (vor dem *pointcut*), *after* (nach dem *pointcut*) und *around* (anstelle des *pointcuts*). Folgendes Beispiel gibt das Wort „XY“ vor der Ausführung des oben genannten *pointcuts* x aus:

```
before(): x() {
  System.out.println( "XY" );
}
```

Abbildung 2.7 verdeutlicht die Funktionsweise von AspectJ anhand einer beispielhaften Implementierung eines *logging*-Features.

Das Programm enthält die Klassen X und Y mit einer Methode *move* und eine Klasse Z mit einer Methode *stay* (Zeilen 1 bis 20 im linken Kasten). Das Programm ruft nacheinander die *move*-Methoden von X und Y auf, abschließend die Methode *stay* von Z (Zeilen 22 bis 26 im linken Kasten). Der Aspekt *logging* definiert zwei *pointcuts* (Zeilen 2 und 3 im rechten Kasten). Der erste greift bei jeder Methode namens *move* ohne Parameter, der zweite bei allen Methoden in Klasse Y und Z. Abschließend wird definiert, dass vor jeder Ausführung der Methode *move* der Text „move-Methode wird ausgeführt.. “ und nach der Ausführung jeder Methode in Y und Z der Text „Methode in Klasse Y oder Z beendet.. “ ausgegeben wird (Zeilen 6 bis 12 im rechten Kasten). Bei der Ausführung des Programms wird also folgendes ausgegeben:

```
move-Methode wird ausgeführt..
move-Methode wird ausgeführt..
Methode in Klasse Y oder Z beendet..
Methode in Klasse Y oder Z beendet..
```

Für weiterführende Informationen wird auf die ausführliche Dokumentation des AspectJ-Teams verwiesen.[7] Mit Hilfe dieser Technik können alle Teile der Lastbalancierungsimplementierung an einer Stelle gekapselt werden.

[7] `http://www.aspectj.org`

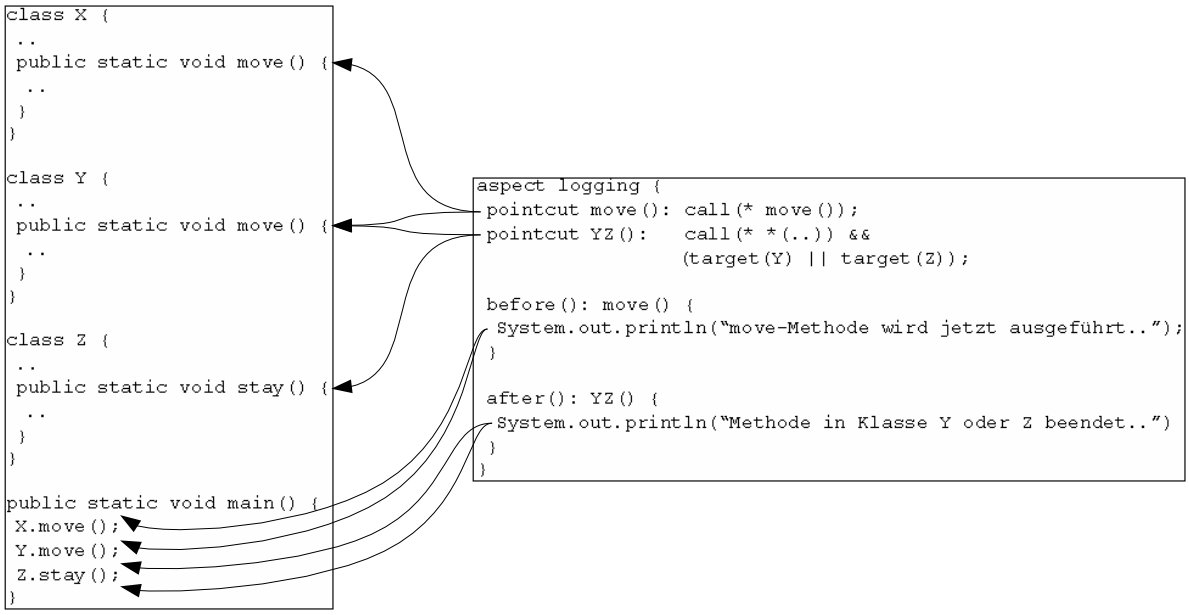

Abbildung 2.7: Logging-Beispiel mit AspectJ

2.3.4 Nutzen von AOP/FOP bei der Implementierung

Aspektorientierte Programmierung gilt als gut geeignet, um Lastbalancierung in verschiedenen Bereichen zu implementieren ([PB02], [Mon04], [CGM99]). In dieser Arbeit wird überprüft, ob das auch für DHTs, insbesondere von DHTs auf Basis von Programmfamilien, gilt. Dabei müssen viele Basisklassen des Protokolls erweitert werden. Mit klassischer objektorientierter Programmierung kann dabei eine Trennung der Spezialfunktion Lastbalancierung vom Kern des Protokolls nicht oder nur sehr schwer erreicht werden. Die P2P-Programmfamilie ist sehr stark auf eine Trennung einzelner Funktionen ausgerichtet, so dass sie sich sehr gut auf spezielle Anforderungen konfigurieren lässt. Mit featureorientierter Programmierung wird die Konfiguration auf einfache Weise durch simple Auswahl der gewünschten *Features* mittels deklarativer Beschreibung ermöglicht. Bei der Implementierung verschiedener Methoden zur Bestimmung der Auslastung von Peers und zur Verteilung von Last sowie deren Kombination unterstützt FOP die Auswahl, welche dieser Methoden genutzt werden sollen. Auch die Nutzung der Lastbalancierung an sich lässt sich auf diese Weise ein- und ausschalten.

Kapitel 3

Die Peer-to-Peer-System Programmfamilie

Die Peer-to-Peer-Programmfamilie wurde mit Hilfe von FOP und AOP erstellt. Sie enthält eine Vielzahl von *Features* – implementiert durch *Mixin Layer* –, die nach fünf verschiedenen Themengebieten klassifiziert und aufgeteilt wurden. Implementiert wurde die Programmfamilie in Java, allerdings mit Hilfe der Erweiterungen ATS und AspectJ, die in Abschnitt 2.3.2 und 2.3.3 beschrieben wurden.

Im diesem Kapitel werden die fünf Themengebiete (P2P Basis, P2P Datenstruktur, P2P dynamische Hashtabelle, das CAN und die Applikationen, vgl. Abbildung 3.1), sowie die drei Teile, die der Unterstützung dienen (globale Aspekte, Dokumentation und Hilfswerkzeuge) näher beschrieben. Jedes Themengebiet ist dabei in einem eigenen Verzeichnis untergebracht (es ist bei den folgenden Kapiteln jeweils in Klammern angegeben), welches wiederum Unterverzeichnisse enthält, die die zugehörigen Layer enthalten.

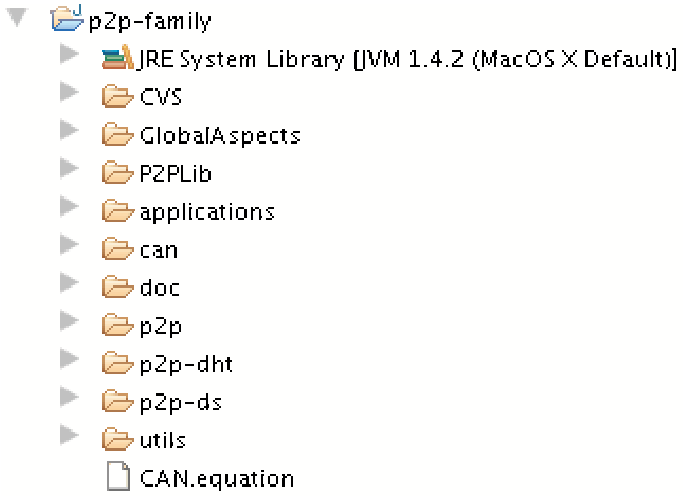

Abbildung 3.1: Verzeichnisstruktur der gesamten P2P-Programmfamilie

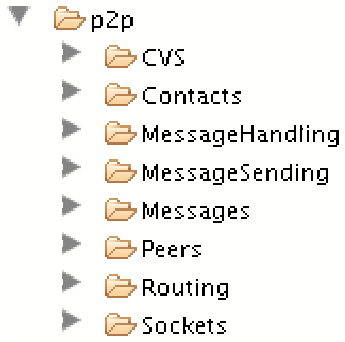

Abbildung 3.2: Verzeichnisstruktur der P2P-Basis

Die CVS-Verzeichnisse in den Screenshots beziehen sich lediglich auf die Quellcodeverwaltung und können daher ignoriert werden.

Zum Zeitpunkt der Erstellung dieser Arbeit enthält die P2P-Programmfamilie 120 Layer und 17 Aspekte. Insgesamt enthält das Projekt etwa 8500 Zeilen Quellcode, der sich auf 124 verschiedene Klassen (Aspekte wurden als Klassen mitgezählt) verteilt.

Es folgt ein Überblick über die wichtigsten Klassen, die in den jeweiligen Layern definiert beziehungsweise erweitert werden. Dieser Auszug ist für das Verständnis der Architektur dieser P2P-Programmfamilie essenziell, um die Implementierung der Lastbalancierung in Kapitel 5 nachvollziehen zu können.

3.1 P2P Basis (p2p)

Im P2P-Verzeichnis wird die Basis für P2P-Systeme definiert (siehe Abbildung 3.2). Dazu gehören Peers, Nachrichten und Nachrichtenverarbeitung, Kontakte, Routing und die Netzwerkkommunikation. Es wird weniger die P2P-Funktionalität implementiert als vielmehr die Struktur des P2P-Systems und die Netzwerkkommunikation über Sockets. Definiert wird

- ein *Peer* mit einer eindeutigen Identifikationsnummer (*PeerId*).
- eine Nachricht (*Message*), die aus einem Nachrichtenkopf sowie einem -körper besteht.
- ein *Listener*, der einen Peer befähigt, auf einem bestimmten Port auf eingehende Nachrichten zu reagieren.
- ein *MessageHandler*, der die Verarbeitung der ankommenden Nachrichten übernimmt.
- ein *MessageSender*, der Nachrichten an andere Peers versendet.

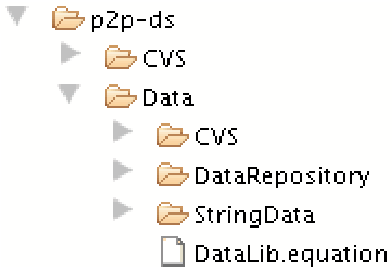

Abbildung 3.3: Verzeichnisstruktur der P2P-Datenstruktur

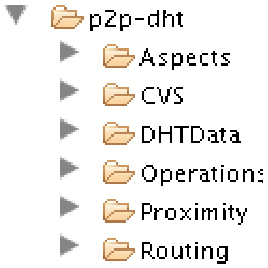

Abbildung 3.4: Verzeichnisstruktur der verteilten Hashtabelle

- eine Kontakteklasse (*Contact*), mit der ein Peer die Rechnernamen und Ports anderer Peers speichern kann.
- eine (leere) Klasse *Routing*, in der später der konkrete Weiterleitungsmechanismus implementiert wird.

3.2 P2P Datenstruktur (p2p-ds)

Im P2P-DS-Verzeichnis (siehe Abbildung 3.3) wird die Grundstruktur für die Datenspeicherung definiert. Konkret sind das:

- ein *DataObject*, das einen String speichert und so jedes serialisierbare Objekt aufnehmen kann.
- das *DataRepository*, in dem später die Datenobjekte *(DataObjects)* gespeichert werden. Diese Speicherung wird hier allerdings noch nicht implementiert.

3.3 Verteilte Hashtabellen (p2p-dht)

Im P2P-DHT-Verzeichnis (siehe Abbildung 3.4) werden die verschiedenen Nachrichtentypen des CAN und deren Verarbeitung, das Routing, die Datenspeicherung, sowie die Verwaltung der Nachbarpeers implementiert. Unterteilt in die wichtigsten Klassen sind das:

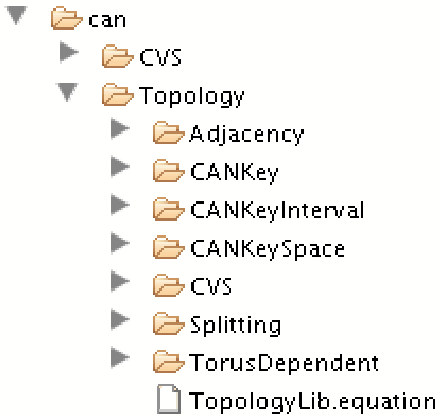

Abbildung 3.5: Verzeichnisstruktur der CAN-Topologie

- Die Klasse *Routing*, die auf Basis der geringsten (euklidischen) Distanz den besten Nachbarn für eine Nachrichtenweiterleitung bestimmt.
- Die Klasse *MessageSender* wird um die Funktion der Nachrichtenweiterleitung erweitert.
- Die Nachrichtenklassen, deren Verarbeiter *(handler)*, und die jeweiligen Bestätigungsnachrichten und deren Verarbeiter für folgende Operationen:
  - Abfragen eines Schlüssel-Wert-Paares
  - Einfügen eines Schlüssel-Wert-Paares
  - Löschen eines Schlüssel-Wert-Paares
  - Einfügen eines neuen Peers in das Netz
  - Hinzufügen/Entfernen von Nachbarn
- Erweiterung der Klasse *Peer* um eine Liste von Nachbarn und Methoden zu deren Verwaltung (implementiert in der Klasse *ImmediateNeighbors*).
- Die Klasse *Key*, die eine Zahl speichert und als Schlüssel der Hashtabelle dient.
- Die Klasse *KeyInterval*, die den minimalen und maximalen Schlüsselwert des P2P-Netzes enthält und so überprüfen kann, ob ein bestimmter Schlüssel zulässig ist.

3.4 CAN-Topologie (can)

Im can-Verzeichnis (siehe Abbildung 3.5) wird das P2P-System so erweitert, dass Schlüssel beliebiger Dimension benutzt werden können. Außerdem wird die Topologie definiert. Dazu werden die folgenden Klassen erweitert:

- Die Klasse *KeyInterval*, die die Nachbarschaft zweier Peers überprüft und die Distanz zwischen zwei Peers berechnet. In diesem Layer wird sie zudem befähigt, mit einem n-dimensionalen Schlüsselbereich umzugehen.

- Das *DataRepository*, das einen Schlüsselbereich in zwei Teile teilen kann (wichtig, wenn neue Peers in das Netz eintreten).

3.5 Zubehör

3.5.1 Applikationen (applications)

Hier finden sich die Applikationen, die auf dem CAN aufbauen. Derzeit ist das ein „Web-Crawler“. Es handelt sich dabei um eine Anwendung, die eine Art Rückwärts-Verlinkungs-Mechanismus zur Verfügung stellt. Die Integration einer kleinen Groupwareanwendung, die bereits existiert, ist geplant.

3.5.2 Globale Aspekte (GlobalAspects)

Alle Aspekte befinden sich in diesem Verzeichnis. Da sie unabhängig von der Konfiguration im Sinne von Layern auf das System angewendet werden, sind sie in einem separaten Verzeichnis zusammengefasst und durch Unterverzeichnisse thematisch gegliedert.

3.5.3 Dokumentation (doc)

In diesem Hilfsverzeichnis befindet sich die Dokumentation des Systems. Allerdings besteht hier noch erheblicher Nachholbedarf. Derzeit werden Konfigurationsaspekte sowie Experimentgenerierung und -durchführung erklärt.

3.5.4 Hilfswerkzeuge (utils)

Das utils-Verzeichnis beinhaltet Skripte und Daten, die nicht direkt zum CAN gehören, aber mit denen sich beispielsweise ein CAN oder fest definierte Experimente im CAN starten lassen, um das Verhalten in verschiedenen Konfigurationen zu testen.

3.6 Ansätze zur Lastbalancierung

Bei der Implementierung der Lastbalancierung wird an mehreren Stellen in die bestehende Programmfamilie eingegriffen. So werden Lastinformationen mit Hilfe der Klasse `MessageSender` (vgl. Abschnitt 3.1) versendet und mit einem `MessageHandler` (vgl.

Abschnitt 3.3) in die Kontaktinformationen der Nachbarpeers (Klasse *Contact*, vgl. Abschnitt 3.1) eingetragen. Um die Last zu verteilen, wird zum Beispiel der Aufteilungsmechanismus des *DataRepository* (vgl. Abschnitt 3.4) zur dynamischen Verteilung von den verwalteten Zonen benutzt. Details zur Implementierung werden in Kapitel 5 vorgestellt.

Kapitel 4

Lastbalancierung in verteilten Hashtabellen

In diesem Kapitel werden zunächst verschiedene Grundbegriffe der Lastbalancierung erarbeitet und definiert. Es folgen eine Klassifizierung und Diskussion der verschiedenen Ziele von Lastbalancierung in P2P-Systemen. Außerdem werden die unterschiedlichen Möglichkeiten, Last zu messen, aufgezeigt und verglichen. Dabei wird insbesondere beschrieben, welche unterschiedlichen Arten von Last existieren, für welche Ziele ihre Messung verwendet werden kann und wo Probleme bestehen. Danach werden mehrere Lastbalancierungalgorithmen aus der Literatur erläutert und analysiert. Abschließend wird der Fokus dieser Arbeit im Bezug auf Lastbalancierung dargestellt sowie eine Auswahl der zu entwickelnden Arten von Last und Lastbalancierungsalgorithmen getroffen.

4.1 Definition der Grundbegriffe

Für verschiedene Begriffe im Bereich Lastbalancierung gibt es keine allgemein akzeptierte Definition (Beispiel: „Last“). Um Unklarheiten zu vermeiden, wurden im Rahmen dieser Arbeit die folgenden Definitionen entwickelt:

Lastart: Als Lastart wird eine bestimmte Art und Weise der Belastung eines Systems bezeichnet. Im CAN sind Beispiele hierfür die Prozessorauslastung, die Anzahl der eingehenden Nachrichten oder die Menge an Daten, die ein Peer speichert.

Last: Der Begriff Last wird in dieser Arbeit im umgangssprachlichen Sinn verwendet, also als Belastung in einer beliebigen Weise. Dies ist erwähnenswert, da zum Beispiel [RPW04] die Lastart „Menge an verwalteten Daten“ synonym zum allgemeinen Begriff Last verwendet. Im Rahmen der Experimente in dieser Arbeit soll überprüft werden, ob diese Annahme haltbar ist. Beispielsweise scheint die Lastart „Prozessorauslastung“ für P2P-Systeme, die auf verteilte Rechenkraft ausgerichtet sind, wesentlich geeigneter. Aber auch im CAN, das verteilte Datenspeicherung

anbietet, können andere Lastarten, wie zum Beispiel die „Anzahl ein- und ausgehender Nachrichten“ ebenfalls ein geeignetes Bild über die Auslastung eines Peers beziehungsweise eines P2P-Systems zeichnen.

Hochlast-Peers: Peers, deren Auslastung über einem definierten Grenzwert liegt, bezeichnet man als Hochlast-Peers.

Niederlast-Peers: Analog spricht man von Niederlast-Peers, wenn ihre Auslastung unterhalb dieses Grenzwerts liegt.

Lastschwelle: Den Grenzwert der Auslastung, der festlegt, ob ein Peer ein Hochlast- beziehungsweise ein Niederlast-Peer ist, nennt man die Lastschwelle.

Lastbalancierung: Unter Lastbalancierung versteht man die Verteilung von Belastung eines Systems auf mehrere seiner Teile. Maßstab für die Verteilung bildet die Last der einzelnen Teile. Die Lastbalancierung im CAN soll dafür sorgen, dass einzelne Peers bei Hochlast durch andere, niedrig(er) belastete Peers entlastet werden, so dass sich die gewünschte Lastverteilung einstellt. Selbst bei gleichmäßiger Verteilung der Anfragen kann eine Verbesserung mit Hilfe von Lastbalancierung erreicht werden, beispielsweise wenn die einzelnen Peers unterschiedlich leistungsfähig sind (was in realen Anwendungen eher die Regel als die Ausnahme sein dürfte).

4.2 Lastbalancierungsziele

Es gibt im wesentlichen drei verbreitete, orthogonale Ziele für die Lastbalancierung in P2P-Systemen:

- **maximale Gesamtperformance**

  Das offensichtlichste Ziel ist es, die Leistung eines P2P-Systems zu maximieren. Mit Leistung ist bei DHTs die Anzahl der Anfragen gemeint, die ein P2P-System innerhalb einer bestimmten Zeit beantworten kann. In P2P-Netzen, deren verteilte Kapazität wie bei dem SETI-Projekt die Rechenleistung ist, misst man die Leistung anhand der Anzahl von durchgeführten Berechnungen. Bei diesem Ziel ist im besonderen darauf zu achten, dass die Implementierung der Lastbalancierung möglichst wenig *Overhead* am eigentlichen Protokoll verursacht, da dieser *Overhead* die Leistung des Systems mindert.

- **Gerechtigkeit durch gleichmäßige Verteilung**

  In vielen Bereichen des menschlichen Zusammenlebens wird darauf Wert gelegt, dass jede (gleichberechtigte) Person die gleiche Menge an Arbeit verrichtet. So kann ökonomisch gesehen Gerechtigkeit erreicht werden. Auf ein P2P-System abgebildet kann das bedeuten, dass die Lastbalancierung das Ziel hat, beispielsweise die Menge der gespeicherten Daten oder die Menge der verarbeiteten Anfragen je Peer

möglichst gleichmäßig zu verteilen. In vielen P2P-Netzen existiert das sogenannte *Freeriding*-Problem ([ABCM04], [KSTT04],[RL03]). Es beschreibt die Eigenschaft, dass nur wenige Peers ihre Ressourcen (wie Prozessorleistung oder bestimmte Dateien) anbieten und sehr viele davon profitieren, ohne wiederum ihre eigenen Ressourcen mit den anderen Peers zu teilen. Bekannte Ansätze zur Bekämpfung dieses Problems sind Reputationsmechanismen ([KSGM03], [BB04a], [DMS03]) und marktorientierte Belohnungs- und Bezahlsysteme ([SD02], [HLM+03]). Eine weitere Möglichkeit zur Vermeidung von *Freeriding* ist es, die Lastbalancierung so zu nutzen, dass jeder Peer den gleichen Anteil der anfallenden Aufgaben übernehmen muss. Es besteht allerdings keinesfalls Einigkeit darüber, ob Freeriding überhaupt ein Problem ist.[1]

- **Gerechtigkeit durch aktivitätsbezogene Lastverteilung**

  Dieses Ziel unterscheidet sich vom Ziel der gleichmäßigen Verteilung dadurch, dass die eigene Aktivität der Peers berücksichtigt wird. So sollen nicht alle Peers die gleiche Arbeit durchführen, sondern überproportional viel Arbeit von Peers, die besonders viele Ressourcen des P2P-Systems beanspruchen, übernommen werden (vgl. [Knö03]). Dadurch kann *Freeriding* noch feingranularer als bei gleichmäßiger Verteilung über alle Peers bekämpft werden.

4.3 Lastarten

Bei der Lastbalancierung in DHTs können verschiedene Lastarten betrachtet werden, um den Grad der Auslastung zu bestimmen. Die in Tabelle 4.1 beschriebenen Lastarten wurden dabei im Rahmen dieser Arbeit erfasst und klassifiziert.

Lastart	Beschreibung/Besonderheiten
(1) Prozessorauslastung	Mit der Prozessorauslastung wird die Auslastung des Hauptprozessors (*central processing unit*, CPU) bezeichnet. Bei mehreren Prozessoren wird dabei die durchschnittliche Auslastung aller Prozessoren betrachtet.
(2) Datenübertragungsrate der Netzwerkschnittstelle	Netzwerkschnittstellen haben eine genau definierte Übertragungskapazität. Derzeit sind das normalerweise zwischen 100MBit und 1GBit für interne Netzwerke und zwischen 56KBit (Modem-Leitungen) und einigen MBit für Internetverbindungen.

[1] `http://www.openp2p.com/pub/a/p2p/2000/12/01/shirky_freeloading.html`

(3) belegter beziehungsweise freier primärer Speicher (Arbeitsspeicher)	Ein zu voller Arbeitsspeicher führt zum Auslagern von Daten auf die Festplatte *(swapping)*, was unter Umständen zu dramatischen Geschwindigkeitseinbußen führen kann.
(4) belegter beziehungsweise freier sekundärer Speicher (Festplattenspeicher)	Eine volle Festplatte kann keine weiteren Daten mehr aufnehmen und führt so zu Fehlern im System. Je voller eine Festplatte ist, desto höher ist die Wahrscheinlichkeit, dass Daten in fragmentierter Form gespeichert sind. Damit verbunden ist ein Performanceverlust bei Lese- und Schreiboperationen.
(5) Menge der von einem Peer verwalteten Daten	Die Datenmenge kann als Größe und/oder als Anzahl der gespeicherten Objekte interpretiert werden.
(6) Anzahl der ein- und ausgehenden Nachrichten	Hier kann zwischen weiterzuleitenden Nachrichten (Zieladresse außerhalb des verwalteten Datenbereichs) und zu verarbeitenden Nachrichten (Zieladresse im verwalteten Datenbereich) unterschieden werden, da a) eine Nachricht, die weitergeleitet werden muss, weniger Aufwand verursacht und b) diese als weniger wichtig betrachtet werden kann, da sie ohne weiteres auch von anderen Peers bearbeitet werden könnte.
(7) Anzahl der von einem Peer ausgehenden Nachrichten	Passend zum Balancierungsziel „Gerechtigkeit durch aktivitätsbezogene Lastverteilung“ wird hier die Auslastung eines Peers anhand der von ihm ausgehenden Anfragen berechnet. Eine hohe Anzahl ausgehender Anfragen meldet ein Peer als geringe Last. Dadurch muss er Arbeit von Peers mit einer sehr geringen Anzahl ausgehender Anfragen (deren Auslastung in diesem Fall als besonders hoch eingestuft wird) übernehmen.

Tabelle 4.1: Beschreibung verschiedener Lastarten

Man kann zwischen physischen und virtuellen Lastarten unterscheiden:

1. **Physische Lastarten**

   Bei physischen Lastarten, die die Auslastung von Hardware wiedergeben, ist es recht einfach, Grenzwerte zu finden, da die Auslastung stets in Prozent der maximalen Leistungsfähigkeit angegeben werden kann. Die Messung physischer Lastarten

– zu ihnen gehören (1), (2), (3) und (4) in Tabelle 4.1 – ist für das Lastbalancierungsziel der maximalen Gesamtperformance eines P2P-Systems geeignet.

2. **Virtuelle Lastarten**

   Zu den virtuellen Lastarten gehören (5), (6) und (7) aus Tabelle 4.1. Sie haben keine feste Obergrenze (abgesehen von den Grenzen der darunter liegenden Hardware). Es ist also komplizierter zu definieren, wann Hochlast anliegt, da dies vom jeweiligen Anwendungsszenario abhängt. Wird eine Gleichverteilung angestrebt, so ist ein Peer schon Hochlastpeer, wenn er nur wenig über der Durchschnittslast der anderen Peers steht. Andererseits können auch die unterschiedlichen Kapazitäten einzelner Peers bei der Definition von Hochlast berücksichtigt werden. Da es kein globales Wissen über entsprechende Zahlen für alle Peers gibt, muss ein Peer sich mit seinen Nachbarn austauschen, um zu bestimmen, ob er im Verhältnis zu den anderen überlastet ist. Dadurch wird in jedem Fall zusätzliche Kommunikation benötigt, die die Effizienz des Systems mindert. Mit Hilfe von Messungen der virtuellen Lastarten lassen sich die beiden Gerechtigkeitsziele („Gerechtigkeit durch gleichmäßige Verteilung“, „Gerechtigkeit durch aktivitätsbezogene Lastverteilung“) verfolgen.

4.4 Methoden zur Lastbalancierung

In [Knö03] beschreibt eine vorhergehende Arbeit die folgenden grundsätzlichen Methoden zur Lastbalancierung, die hier zusätzlich anhand grafischer Beispiele verdeutlicht werden.

4.4.1 Zonentransfer

Dieses Konzept wurde 2001 von Dabek et al. in [DKK+01] als Prinzip der *Virtual Servers* vorgestellt. Es setzt voraus, dass jeder Peer mehrere Zonen verwaltet. Er enthält gewissermaßen virtuelle Peers mit eigener Nachbarverwaltung. Abbildung 4.1 zeigt, dass ein Niederlast-Peer auf diese Weise einem Hochlast-Peer die Verwaltung von einem oder mehreren virtuellen Peers abnehmen und so dessen Auslastung senken kann. Peer 1 und Peer 2 verwalten dort jeweils vier virtuelle Peers. Da Peer 1 unter Hochlast steht, gibt er die Verwaltung eines virtuellen Peers (und damit dessen Datenbereich) an Peer 2 ab. Dadurch wird Peer 1 entlastet.

Allerdings steigt der Verwaltungsaufwand jedes Peers stark an. Für jeden virtuellen Peer müssen Nachbarn gepflegt werden und Nachrichten durchlaufen viel mehr (virtuelle) Peers bis zum Ziel. Dabei wird jedesmal durch den Routingalgorithmus eine Berechnung ausgeführt sowie eine Netzwerkverbindung zum ermittelten virtuellen Peer aufgebaut.

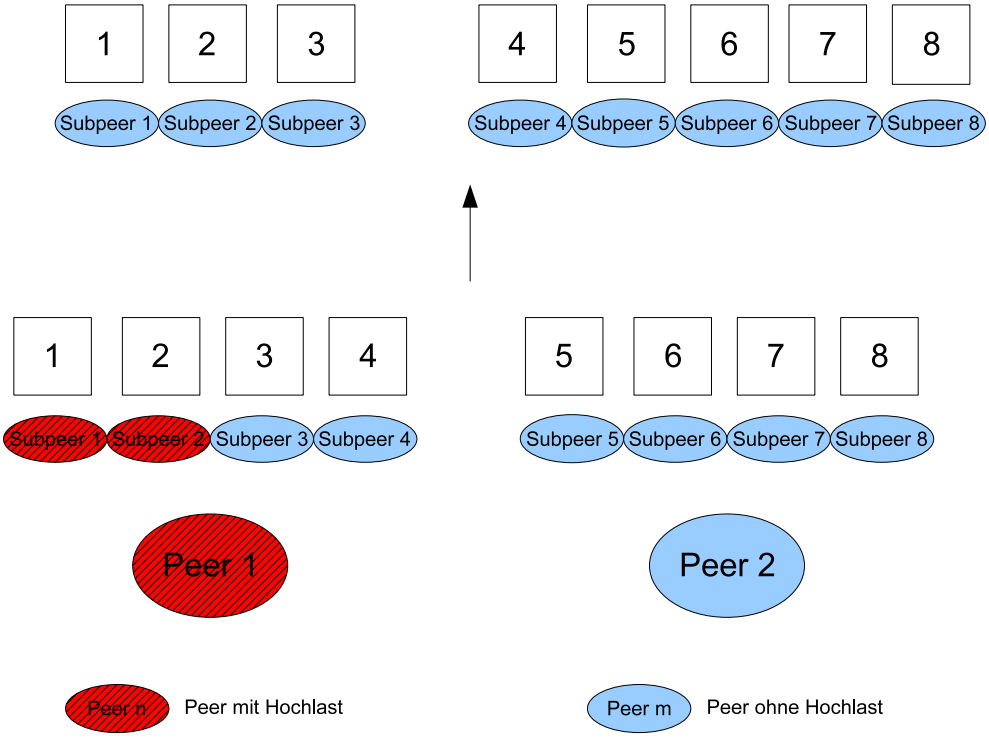

Abbildung 4.1: Lastbalancierung durch Zonentransfer

4.4.2 Replikation

Replikationsmechanismen sind weit verbreitete Ansätze zur Lastbalancierung in verteilten Systemen (unter anderem bei [VBW98], [LKO+00] und [NOT02]). Im Gegensatz zum Zonentransfer werden hier nicht mehrere Zonen bei einem Peer, sondern eine Zone von mehreren Peers verwaltet. Anstatt jeden Wertebereich von einem Peer verwalten zu lassen, helfen eines oder mehrere Replikate von besonders belasteten Bereichen, die Gesamtperformance zu verbessern. Dabei werden die Anfragen an einen bestimmten Datenbereich auf die Replikate verteilt, so dass jedes einzelne Replikat weniger Anfragen beantworten muss. Abbildung 4.2 verdeutlicht dieses Prinzip. Die Nachfrage nach Daten in den Datenbereichen 1 und 2 ist so groß, dass die Peers 1 und 2 unter Hochlast stehen. Peer 3 wird nun zusätzliches Replikat dieser Datenbereiche. Durch die Verteilung der Anfragen werden Peer 1 und 2 entlastet. In der Literatur sind verschiedene Algorithmen zur Anfragenverteilung bekannt (vgl. [Knö03]):

- So beschreibt [GC00] die Verteilung anhand der absoluten Last der Peers,
- [CC97] und [ZYZ+98] hingegen propagieren eine Verteilung anhand der freien Kapazität,
- während [RB02] vorschlägt, Anfragen anhand der Gesamtkapazität der Replikate zu verteilen.

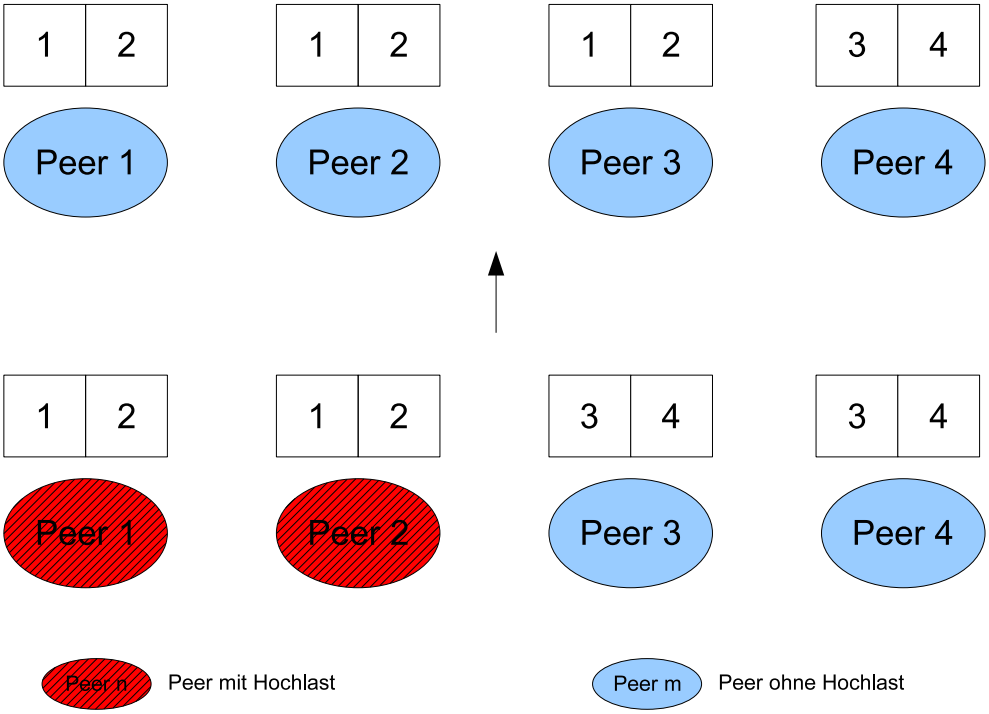

Abbildung 4.2: Lastbalancierung durch Replikation

Kann man dafür sorgen, dass jeder Wertebereich von mindestens zwei Peers verwaltet wird, erhöht sich zudem automatisch die Fehlertoleranz, da bei einem Ausfall eines Peers keinerlei Daten verloren gehen. Allerdings hat auch dieses Verfahren einige Nachteile. So wird durch die Redundanz ein deutlich höheres Gesamtvolumen an Daten verwaltet. Eine weitere Erhöhung des Aufwands entsteht bei der Verwaltung der Nachbarn, da auch deren Replikate gepflegt werden müssen. Der zusätzliche Aufwand und die Erhöhung des verwalteten Datenvolumens sind dabei proportional zum Replikationsgrad. Eine Entlastung einzelner Peers kann zudem nur bei der Abfrage von Daten erreicht werden, nicht bei Einfügeoperationen.

Die Implementierung der Replikation beinhaltet zwei Schritte:

1. **Verteilung der Anfragen auf die Replikate**

   Da alle Replikate die Anfragen für ihre Zone beantworten können, soll durch eine möglichst geschickte Verteilung der Anfragen die bestmögliche Gesamtperformance erreicht werden.

2. **Dynamisches Anlegen und Vernichten von Replikaten**

   Im laufenden Betrieb soll das CAN dazu in der Lage sein, Replikate von gering belasteten Zonen an höher belastete Zonen abzugeben, ebenfalls mit dem Ziel, die Gesamtperformance zu erhöhen. Dazu sendet ein Hochlastpeer eine Anfrage an einen weniger belasteten Peer. Befindet sich dieser in einer Zone mit ausreichend vorhandenen Replikaten, so verwirft er seine bisherigen Daten, übernimmt

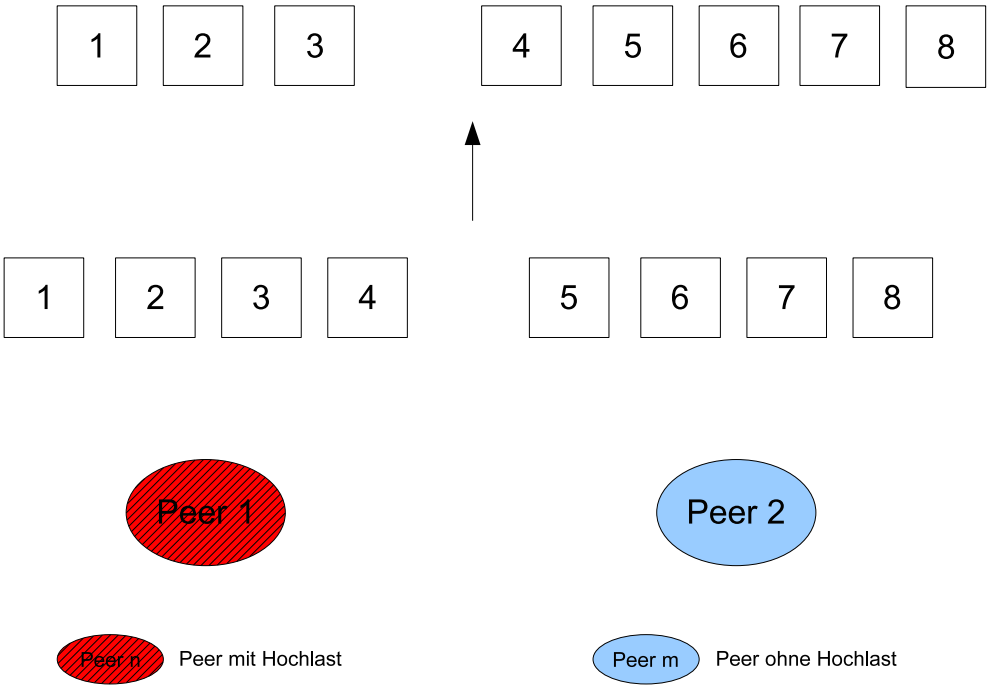

Abbildung 4.3: Lastbalancierung durch Zonenreorganisation

diejenigen des Hochlastpeers und fungiert als zusätzliches Replikat von dessen Datenbereich.

4.4.3 Zonenreorganisation

Eine weitere Methode besteht darin, abhängig von der Auslastung der Peers den Wertebereich, den diese verwalten, zu verändern. Hochlast-Peers werden entlastet, indem man den von ihnen verwalteten Wertebereich verkleinert und den „abgeschnittenen“ Bereich an einen Wertebereich eines Niederlast-Peers übergibt. Zudem wird der Algorithmus für den Netzaufbau dahingehend verändert, dass neue Peers ihre Arbeit nicht in einem zufälligen Bereich aufnehmen, sondern in hoch belasteten Zonen. Bei der Zonenreorganisation wird zwar die Last aller Operationen balanciert, allerdings muss Fehlertoleranz/Ausfallsicherheit noch zusätzlich implementiert werden.

4.4.4 Kombination von Replikation und Zonenreorganisation

Weitere Verbesserungsmöglichkeiten verspricht sich Knöfel in [Knö03] durch die Kombination von Zonenaufteilung und Replikation. Dabei werden für die Anzahl der Replikate Ober- und Untergrenzen festgelegt. Erreichen Niederlastzonen die Replikatsuntergrenze, so können sie sich mit angrenzenden Niederlastzonen vereinigen, um wieder Replikate freigeben zu können. Bei Erreichen der Obergrenze für die Anzahl der Replikate wird analog dazu eine Zone aufgeteilt, um jeweils die Aufnahme neuer Peers zu ermöglichen.

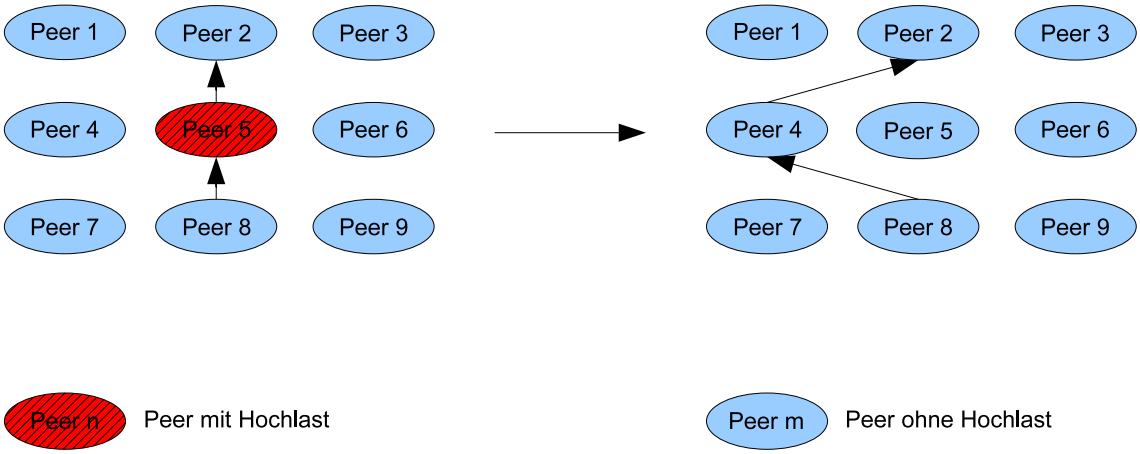

Abbildung 4.4: Lastbalancierung durch lastabhängiges Routing

4.4.5 Lastabhängiges Routing

Als letzte Methode zur Lastbalancierung wird das lastabhängige Routing propagiert. Abbildung 4.4 zeigt, dass dabei Hochlastpeers durch eine geschickte Umleitung von dem Aufwand, Nachrichten weiterzuleiten, entlasten werden. Ohne einen solchen Mechanismus sind Hochlastpeers durch das Routing auf besondere Weise belastet, da sie durch ihre hohe Aktivität auch oft im Routingcache von anderen Peers sind. Auch hier existieren zwei Möglichkeiten der Implementierung:

1. **Priorisierung von Nachrichten**

   Routingnachrichten werden niedriger priorisiert, da sie auch von anderen Peers übernommen werden können. Unter Hochlast kann ein Peer das Routing auch explizit ablehnen.

2. **Modifikation der Entfernungsberechnung**

   Normalerweise wird der Nachbar Ziel einer Nachricht, dessen Zone der Zielzone am nächsten ist. Multipliziert man die Distanz mit der Auslastung der Nachbarpeers, können Hochlastpeers von Routingnachrichten befreit beziehungsweise entlastet werden.

4.5 Fazit / Fokus / Ziele

Verschiedene Arbeiten haben sich schon mit Lastbalancierung in P2P-Systemen allgemein beziehungsweise in verteilten Hashtabellen beschäftigt ([BCM03], [RLS+03], [GLS+04], [HW04], [SMK+01], [BKM05]). Dabei wurden verschiedene Algorithmen zur Lastbalancierung entwickelt und sowohl für verschiedene Implementierungen von P2P-Netzen als auch für statische und dynamische Umgebungen implementiert. Eine repräsentative Auswahl davon wurde in diesem Kapitel diskutiert und verglichen. Allerdings

hat sich keine der genannten Arbeiten mit einer flexiblen Implementierung dieser Algorithmen beschäftigt. Der Fokus der vorliegenden Arbeit besteht in einer separierten und modularen Implementierung der Lastbalancierung mit FOP und AOP, die viele Vorteile verspricht. So kann ein P2P-System, bei dem verschiedene Algorithmen auf diese Weise implementiert wurden, sowohl bei der Programmgenerierung als auch dynamisch zur Laufzeit sehr leicht darauf konfiguriert werden, bestimmte Algorithmen davon zu nutzen, um spezifischen Anforderungen gerecht zu werden. Außerdem kann jede Implementierung als Basis für ein anderes P2P-System dienen, so dass nur wenig Änderungen notwendig sind. Bei klassischer objektorientierter Programmierung würde eine Lastbalancierungsimplementierung hauptsächlich aus der untrennbaren Erweiterung des bestehender Kerns bestehen, so dass eine Portierung auf ein anderes System sehr mühselig und kostenintensiv wäre.

Um zu zeigen, welche Vorteile sich durch den Einsatz von FOP und AOP im Bezug auf Konfigurierbarkeit und Wiederverwendung bieten, werden in dieser Arbeit jeweils zwei der vorgestellten Algorithmen und Lastmessungsmethoden implementiert. Dabei erfolgt die Auswahl unabhängig von erwarteten Ergebnissen. Anhand der in Kapitel 6 entwickelten Experimente kann allerdings jederzeit überprüft werden, welcher Algorithmus für welches Anwendungsszenario am besten geeignet ist. Auch Konfigurationen mit mehreren parallel aktivierten Lastbalancierungsalgorithmen oder Lastmessungsmethoden können leicht erstellt werden. Implementiert werden

- ein Replikationsmechanismus.
- ein Zonenreorganisationsmechanismus.
- sowie die Messung von
  - Prozessorlast.
  - der Anzahl der ein- und ausgehenden Nachrichten.

Kapitel 5

Implementierung

Dieser Abschnitt beschreibt die Realisierung der Lastbalancierung in der in Kapitel 3 vorgestellten P2P-Programmfamilie. Die Grundlage dafür bildet die Entwicklung eines Mechanismus zur Metadatenverteilung, im speziellen zur Lastdatenverteilung. Damit wird ein verteilt koordiniertes, globales Wissen über Lastinformationen erreicht. Erst dadurch können die Lastverteilungsmethoden bestimmen, welche Peers Hochlastpeers am besten unterstützen können, nämlich diejenigen, die selber am wenigsten belastet sind. Auch zur Messung virtueller Lastarten sind diese Informationen essentiell, da es keine absolute, sondern nur eine zu anderen Peers relative Last gibt. [BA05] beschreibt Charakteristika und Ansätze zur Metadatenverteilung. Diese werden hier so angepasst, dass nur die Nachbarn eines Peers dessen Lastinformationen benötigen, dafür mit dem geringst möglichen Aufwand (Beschreibung in Abschnitt 5.1).

Danach werden die Ziele der Entwicklung einer flexiblen und erweiterbaren Architektur für modulare und konfigurierbare Implementierungen von Lastbalancierungsmechanismen vorgestellt (Abschnitt 5.2). Wie diese Ziele erreicht werden, wird ebenfalls in diesem Abschnitt erläutert. Es folgen abschließend Details zur Implementierung der Lastmessungen (Abschnitt 5.3) und der Methoden zur Balancierung der Last (Abschnitt 5.4). Dabei wird jeweils auf mögliche Entwurfsentscheidungen und die zu konfigurierenden Parameter (wie zum Beispiel die maximale Anzahl von Replikaten in einer Zone) eingegangen.

5.1 Verteilung von Lastdaten

Ein Peer benötigt Lastinformationen von seinen Nachbarn. Steht er unter Hochlast, so kann er eine Lastbalancierung durch eine der in Abschnitt 4.4.1 bis Abschnitt 4.4.5 beschriebenen Methoden anstoßen. Dabei wählt er den Peer mit der geringsten Belastung aus, um diesem eine Balancierungsanfrage zu schicken. Allerdings müssen stets aktuelle Lastinformationen der Nachbarpeers verfügbar sein. Eine andere Möglichkeit besteht darin, Lastinformationen nicht periodisch auszutauschen, sondern erst, wenn ein Peer unter Hochlast steht. Dadurch werden im allgemeinen weniger Nachrichten ausgetauscht, der

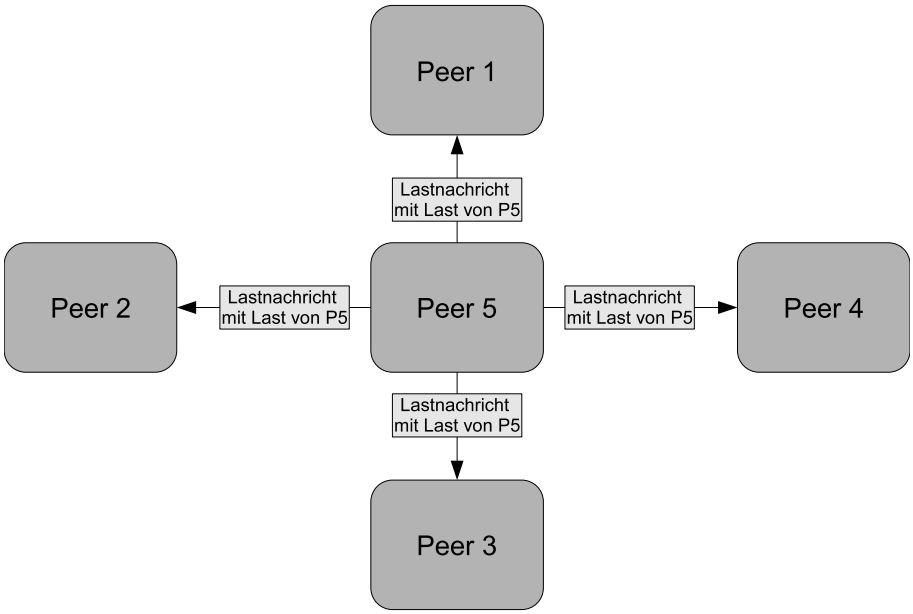

Abbildung 5.1: Aktive Lastdatenverteilung

Vorgang der Lastverteilung dauert jedoch länger. Da ein unter Hochlast stehender Peer so schnell wie möglich entlastet werden soll, wird hier die erste Variante implementiert.

Es gibt zwei prinzipielle Möglichkeiten, Lastdaten zu verteilen:

1. **Aktive Lastdatenverteilung**

   Bei der aktiven Lastdatenverteilung werden Lastdaten wie andere Nachrichten verteilt (vgl. Abbildung 5.1). Periodisch verschickt jeder Peer seine aktuelle Last als Nachricht an seine Nachbarn. Analog zum bisher im CAN verwendeten Prinzip kann der Austausch auch über eine Anfrage mit Antwort geschehen. Allerdings ist es relativ aufwendig, Nachrichten zu verschicken, da eine TCP-Verbindung aufgebaut werden muss. Daher kann etwa die Hälfte des Aufwands vermieden werden, wenn auf die in diesem Fall nicht notwendigen Antwortnachrichten verzichtet wird.

2. **Passive Lastdatenverteilung**

   Von passiver Lastdatenverteilung (engl.: *lazy dissemination*) spricht man, wenn Lastdaten durch andere Nachrichten verteilt werden. Dabei wird einer Nachricht, die versendet werden soll, die Lastinformation des aktuellen Peers angehängt (vgl. Abbildung 5.2). Beim Empfang einer Nachricht kann diese Information dann ausgewertet werden. Buchmann und Apel beschreiben in [BA05] eine flexible Implementierung im CAN und Strategien zur passiven Verteilung von Metadaten (zu denen auch Lastdaten gehören). Dabei werden die Metadaten durch einen Aspekt an zu sendende Nachrichten angehängt. Mehrere *Mixin Layer* implementieren die unterschiedlichen Strategien zur Verteilung dieser Metadaten. Für den speziellen

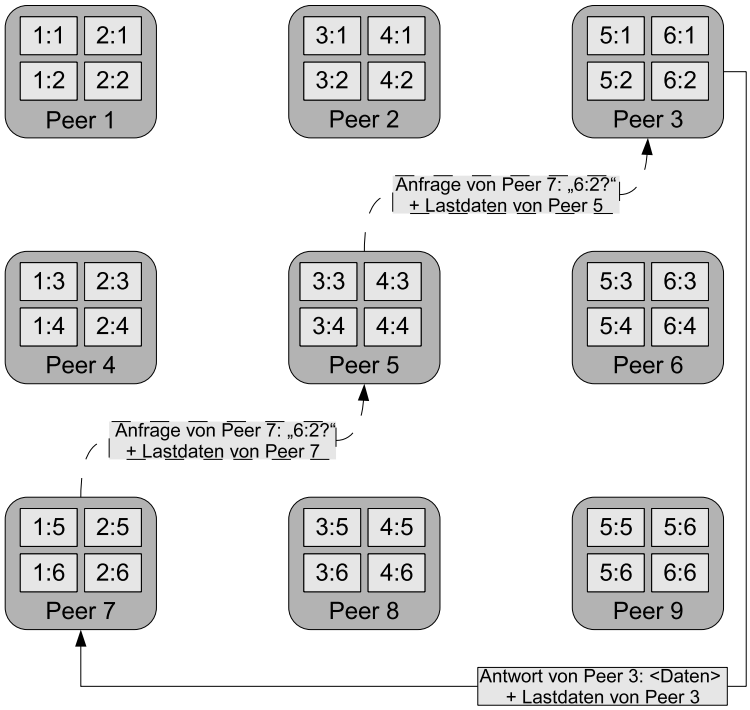

Abbildung 5.2: Passive Lastdatenverteilung

Fall der Lastbalancierung genügt eine recht einfache Strategie. Die Lastinformationen werden nur zu den eigenen Nachbarn weitergeleitet. Jeder Peer entnimmt sie ankommenden Nachrichten und trägt stattdessen seine eigenen ein, falls die Nachricht weitergeleitet wird. Um den *Overhead* bei hoch aktiven P2P-Systemen zu verringern, kann über einen Parameter bestimmt werden, wie viel Zeit zwischen zwei Nachrichten vergangen sein muss, damit die Lastinformationen erneut an eine Nachricht angehängt werden.

Um die Vorteile der passiven Lastdatenverteilung zu nutzen und trotzdem stets auf aktuelle Lastinformationen zugreifen zu können, werden die beiden beschriebenen Methoden auf folgende Weise kombiniert:

- Lastinformationen werden entsprechend dem gewählten Zeitintervall an ausgehende Nachrichten angehängt und von den Nachbarn ausgewertet (passive Lastdatenverteilung).

- Bekommen Peers von einem ihrer Nachbarn innerhalb einer bestimmten Zeitspanne keine Nachrichten (und damit auch keine Lastinformationen), so werden separate Nachrichten, die nur Informationen über die aktuelle Last des Peers enthalten, ausgetauscht (aktive Lastdatenverteilung).

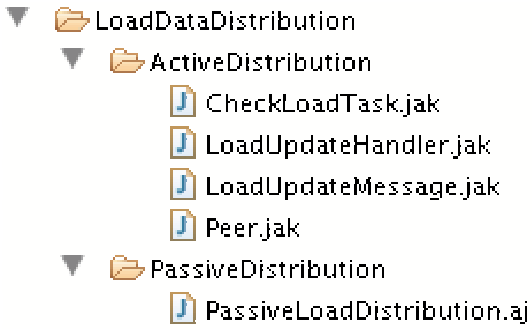

Abbildung 5.3: Übersicht der Implementierung der Lastdatenverteilung

Abbildung 5.3 zeigt die beiden *Layer*, in denen die aktive beziehungsweise passive Lastdatenverteilung implementiert wird. Beim passiven Austausch *(PassiveDistribution)* genügt ein Aspekt, um einerseits abgehenden Nachrichten die Information über die aktuelle Last anzuhängen und andererseits diese Information bei eingehenden Nachrichten zu verarbeiten. Bei der der aktiven Lastdatenverteilung *(ActiveDistribution)* startet ein *Timer* regelmäßig den *CheckLoadTask*. Dieser überprüft in einem definierten Zeitintervall, wann welcher Nachbar zuletzt über die eigene Last informiert wurde. Wird eine bestimmte Grenze überschritten, so wird eine Nachricht vom Typ *LoadUpdateMessage* geschickt, die der Empfänger dann mit dem *LoadUpdateHandler* verarbeitet. Dort wird einfach die Lastinformation des Senders aktualisiert.

5.2 Architektur

Die Lastbalancierung wird mit objektorientierter Programmierung realisiert. Der Einsatz aspektorientierter und featureorientierter Erweiterungen hilft, die Architektur flexibel zu gestalten. Konkret wird Java als Basis-Programmiersprache, AspectJ für AOP und die ATS für FOP eingesetzt. Einerseits handelt es sich dabei um an der Universität Magdeburg bewährte Werkzeuge, andererseits wurde die bisherige P2P-Programmfamilie mit diesen Mitteln implementiert, so dass sich die Implementierung der Lastbalancierung gut integrieren kann.

5.2.1 Ziele der Architektur

Die zu entwickelnde Architektur soll dabei in zweierlei Hinsicht Flexibilität bieten:

- **Erweiterbarkeit/Wiederverwendbarkeit**

  Die Messung einer neuen Lastart oder die Realisierung einer neuen oder anders implementierten Lastbalancierungsmethode soll möglichst keine Redundanz in Programmieraufwand und Quellcode mit sich bringen.

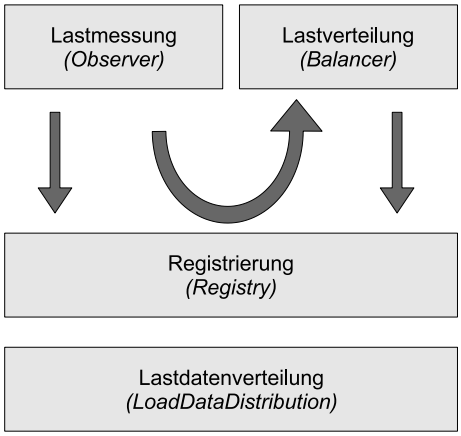

Abbildung 5.4: Grundlegendes Konzept der Architektur

- **Flexibilität/Konfigurierbarkeit**

  Idealerweise wird die Nutzung der verschiedenen Methoden zur Lastmessung und -balancierung in der Programmfamilie mit einer einfachen deklarativen Datei wie sie in Abschnitt 2.3.2 beschrieben wird, konfiguriert. Analog zur bereits existierenden Programmfamilie soll auf diese Weise eine einfache Konfigurierbarkeit beziehungsweise der flexible Einsatz unterschiedlicher Methoden gewährleistet werden. Allerdings muss auf die Besonderheiten der Lastbalancierung wie zum Beispiel den Einsatz von nativen Bibliotheken eingegangen werden.

5.2.2 Entwicklung der Architektur

Das grundlegende Konzept der Architektur wird in Abbildung 5.4 dargestellt. Die Struktur ihrer Implementierung im Dateisystem zeigt Abbildung 5.5. Um das Ziel der Wiederverwendung zu erreichen, werden abstrakte Basisklassen für Lastmessungen *(Observer)* und Lastverteiler *(Balancer)* sowie deren Methoden in einem Basis-*Layer BalancingBase* definiert. Eine Erweiterung um eine neue Lastmessung muss nur die eigentliche Messung implementieren. Lastschwelle und Dauer der Messungen werden bei der Initialisierung der Messung im Aspekt *LoadBalancing* angegeben. Dieser ist dafür zuständig, die Messungen und die Balancierer zu initialisieren und zu verknüpfen. Meldet eine Messung Hochlast, so wird über die Registrierung automatisch die passende Lastverteilung gestartet. Analog wird eine neue Methode zur Lastbalancierung von *Balancer* abgeleitet und implementiert nur die Lastverteilung. Allerdings können auch neue Aspekte oder native Bibliotheken nötig sein, um die Funktionalität der *Observer* und der *Balancer* zu implementieren. So fügt ein Aspekt für die in Abschnitt 5.3.2 beschriebene Methode jedem Peer einen Nachrichtenzähler hinzu, der sich beim Empfang beziehungsweise Versand einer Nachricht erhöht. Sowohl Aspekte als auch native Bibliotheken können nicht durch die ATS verarbeitet werden. Wegen besserer Übersicht werden sie dennoch in den Verzeichnissen abgelegt, in denen der zugehörige *Layer* implementiert wird.

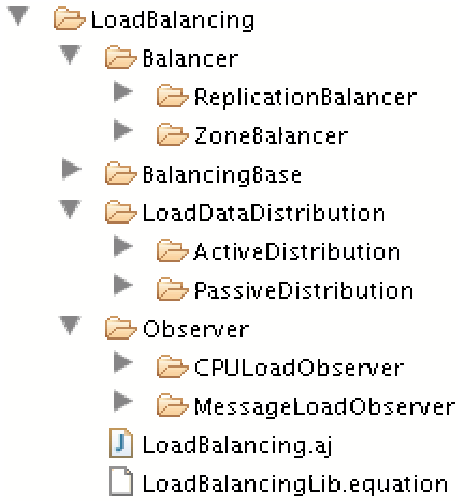

Abbildung 5.5: *Layer*-basierter Aufbau der Lastbalancierung

Um flexibel verschiedene Konfiguration erstellen zu können, wird die ATS (beschrieben in 2.3.2) eingesetzt. Obwohl die AspectJ-Erweiterungen nicht in der ATS integriert sind, können Funktionen auf diese Weise hinzugefügt beziehungsweise entfernt werden. Allerdings müssen die dazugehörigen Aspekte sowie die Initialisierung und Zuordnung der Klassen im koordinierenden Aspekt auskommentiert werden. Auch damit sind Veränderungen noch relativ leicht möglich. Mit ATS-Methoden erweiterbare Aspekte würden jedoch die Handhabung unterschiedlicher Konfigurationen noch weiter vereinfachen. Dabei handelt es sich aber nicht um ein konzeptionelles, sondern nur um ein technisches Problem. Einen integrierten Ansatz von AOP und FOP, allerdings für die Programmiersprache C++, zeigen Apel et al. in [ALRS05].

Eine Registrierung erlaubt die flexible Zuordnung von Methoden zur Lastmessung und Balancierung. Jede von *Observer* oder *Balancer* abgeleitete Klasse registriert sich dazu während der Ausführung des Konstruktors bei der statischen Klasse *Registry*. Der Aspekt *LoadBalancing*, der auch die *Observer* und *Balancer* erstellt, verknüpft sie dann nach Wunsch mit der Methode *assign()* der Klasse *Registry*.

Abbildung 5.6: Übersicht über die Basis-*Mixins*

Der Basis-*Layer* *BalancingBase* ist Bestandteil aller Konfigurationen der P2P-Familie, die Lastbalancierung beinhalten. Er enthält die in Abbildung 5.6 zu sehenden drei *Mixins*, die im folgenden beschrieben werden:

1. ***Registry***

```
layer BalancingBase;

 public abstract  class Observer extends Thread{
     public int currentLoad=-1;
     private int seconds=30;
     private int loadborder=30;
     private Peer m_peer;

     public Observer(Peer peer, int time, int border) {
         m_peer=peer;
         seconds=time;
         loadborder=border;
         this.start();
     }

     public void run() {
         Registry.addLoadObserver(this);
         while( ! isInterrupted() ) {
             currentLoad=getLoad(seconds);
             if(currentLoad > loadborder)
                 Registry.startBalance(this);
         }
     }

     public abstract int getLoad(int time);
}
```

```
layer BalancingBase;

public abstract class Balancer {
    public Peer m_peer;

    public Balancer(Peer peer) {
        m_peer=peer;
        Registry.addBalancer(this);
    }

    public void balance(Observer lo) {
    }

}
```

```
layer BalancingBase;

import java.util.*;

public class Registry {
    public static Vector loadobservers = new Vector();
    public static Vector balancers = new Vector();
    public static Hashtable assignments = new Hashtable(10);
    public static boolean addLoadObserver(Observer lo) {
        return loadobservers.add(lo);
    }

    public static boolean addBalancer(Balancer bal) {
        return balancers.add(bal);
    }

    public static void assignObserverToBalancer(Observer observer, Balancer balancer) {
        String observer_name = observer.getClass().getName();
        String balancer_name = balancer.getClass().getName();
        if(assignments.containsKey(observer_name)) {
            if((assignments.get(observer_name).toString().indexOf(balancer_name) == -1)
                && balancers.contains(balancer))
                    assignments.put(observer_name, balancer_name + " " + (String)assignments.get(observer_name));
        } else
            assignments.put(observer_name, balancer_name);
    }

    public static boolean startBalance(Observer observer) {
        Balancer balancer;
        String observer_name = observer.getClass().getName();
        if(!assignments.containsKey(observer_name))
            return false;
        for(int i=0; i < balancers.size(); i++) {
            balancer=(Balancer)balancers.get(i);
            if(assignments.get(observer_name).toString().indexOf(balancer.getClass().getName()) != -1 )
                balancer.balance(observer);
        }
        return true;
    }
}
```

Abbildung 5.7: Quellcode der Basis-*Mixins*

In der Registrierung – implementiert durch die Klasse *Registry* (siehe Abbildung 5.7 unten) – melden sich die im folgenden beschriebenen Lastmesser und -verteiler bei ihrer Instanzierung an (Zeilen 9-15). Außerdem ordnet der Hauptaspekt *LoadBalancing* mit der Methode *Registry.assignObserverToBalancer()* (Zeilen 17-26) zu, welche Variante der Lastmessung mit welcher Lastverteilung verknüpft wird. Wenn durch eine Lastmessung Hochlast festgestellt wird, so startet diese die Methode *Registry.startBalance()* (Zeilen 28-39), in der die Registrierung die Methode *balance()* (Zeile 36) des passenden Lastverteilers aufruft.

2. ***Observer***

   Die Klasse *Observer* (siehe Abbildung 5.7 links) implementiert den Teil, der für jede Methode zur Lastmessung notwendig ist. Sie erbt von einem *Thread* (Zeile 3), so dass die Messungen immer parallel zur eigentlichen Funktion der Peers laufen können. Beim Start der Lastmessung wird die Verbindung zur Registrierung hergestellt (Zeile 17). Außerdem wird das Ergebnis der regelmäßigen Lastmessungen überprüft, um beim Überschreiten der Lastschwelle die Registrierung anzuweisen, die entsprechenden Lastverteiler zu starten (Zeilen 18-22). Jede Lastmessungsvariante muss von *Observer* abgeleitet werden und nur die Methode *getLoad()* (Zeile 25) implementieren, die die Auslastung eines Peers für eine gegebene Zeitspanne berechnet und zurück gibt.

3. ***Balancer***

   Die Lastverteilung findet in den Klassen statt, die von *Balancer* (siehe Abbildung 5.7 rechts) abgeleitet sind. Sie bietet die Grundstruktur, also die Verbindung mit der Registrierung (Zeile 8), sowie den Prototypen der Methode *balance()* (Zeile 11). Nur diese muss von den Verteilerklassen implementiert werden. Wie im Abschnitt 5.4 gezeigt wird, besteht der größte Aufwand allerdings darin, die Fähigkeit zur Lastverteilung in der P2P-Familie zu implementieren.

Abbildung 5.8 visualisiert die grundsätzliche Struktur der erstellten Klassen/Aspekte und ihre Beziehungen zueinander. Der *LoadBalancing*-Aspekt initialisiert vor dem eigentlichen Start des Peers die konkreten Lastmesser (je ein *Thread*) und -balancierer *(CPULoadObserver, ReplicationBalancer,..)*. Deren Basisklassen implementieren die Funktion zum Anmelden bei der Registrierung. Zur Lastmessung können unterstützende Komponenten wie native Bibliotheken oder Aspekte notwendig sein. Meldet ein Lastmessungsthread, dass der beobachtete Peer unter Hochlast steht, teilt er das der Registrierung mit. Diese startet dann den/die von dem *LoadBalancing*-Aspekt bei der Initialisierung zugeordneten Lastverteiler. In der Abbildung nicht enthalten sind die Klassen aus dem CAN, die zum Teil auf die Besonderheiten der Lastbalancierung angepasst werden mussten. Details dazu werden in den folgenden Abschnitten erläutert.

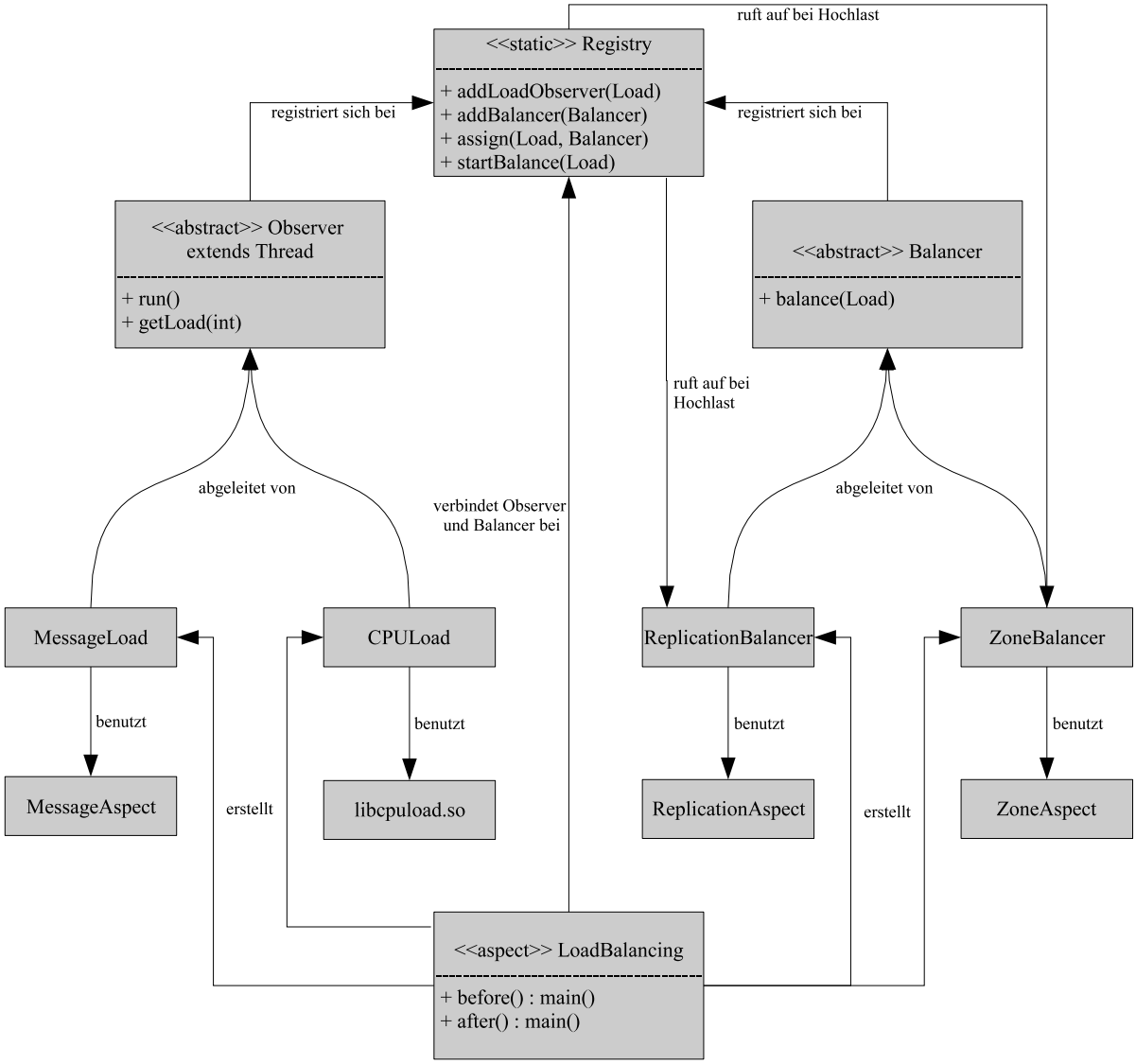

Abbildung 5.8: Grundsätzliches Klassendiagramm der Lastbalancierung

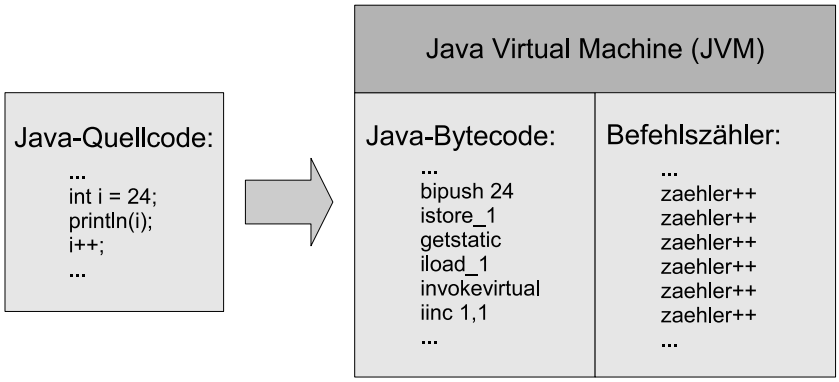

Abbildung 5.9: Abschätzung der Prozessorlast durch Zählung der in der JVM ausgeführten Bytecodebefehle

5.3 Implementierung konkreter Lastmessungen

In diesem Abschnitt wird die prototypische Implementierung der beiden ausgewählten Methoden zur Lastmessung – Messung der Prozessorlast sowie Messung der Anzahl ein- und ausgehender Nachrichten – besprochen. Dazu gehören Diskussionen über verschiedene Möglichkeiten zum Entwurf und welche Parameter eingeführt werden müssen.

5.3.1 Messung der Prozessorlast

Da Java Quellcode nicht in Maschinencode übersetzt, ist ein direkter Hardwarezugriff nicht möglich. Mit Hilfe einer virtuellen Maschine (*JVM, Java Virtual Machine*) wird der vom Java-Compiler erzeugte Pseudomaschinencode ausgeführt. Um die Auslastung des Prozessors dennoch zu messen, gibt es zwei Ansätze:

- **Abschätzung durch die Zählung der von der JVM ausgeführten Befehle**

  In [BHV01] beschreiben Binder et al., wie man die Prozessorauslastung abschätzen kann, indem die ausgeführten Befehle in der JVM gezählt werden (siehe Abbildung 5.9). Das hat den Vorteil, dass diese Methode ohne Änderungen auf allen von Java unterstützten Plattformen lauffähig ist. Leider gibt es zwei erhebliche Nachteile:

  - **Ungenauigkeit**

    Mit dieser Methode kann die Leistungsfähigkeit der Maschine nicht ermittelt werden, so dass bei einem P2P-System mit unterschiedlich leistungsfähigen Peers keine vergleichbaren Messwerte produziert werden. In diesem Fall kann so nur eine gleichmäßigere Verteilung der Last erreicht werden. Dieses Ziel wird aber zum Beispiel mit der in Abschnitt 5.3.2 beschriebenen Methode auf einfacherem und direkterem Weg erreicht.

  - **Großer *Overhead***

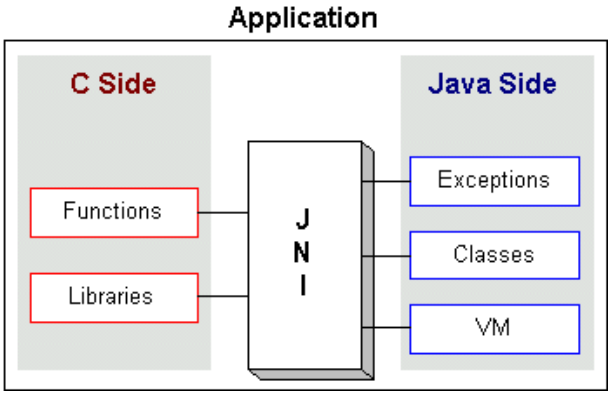

Abbildung 5.10: Verbindung nativer Komponenten mit Java durch das JNI (aus [CWH98])

In [BHV01] wird ebenfalls beschrieben, dass diese Methode einen *Overhead* von deutlich über 10% erzeugt. Dadurch wird das System in etwa um diesen Prozentsatz weniger leistungsfähig. Das Ziel der Lastbalancierung durch die Messung der Prozessorauslastung besteht aber gerade in der Erhöhung der Leistungsfähigkeit. Eine weniger aufwendige Messung ist also wünschenswert.

- **Zugriff auf eine native Bibliothek außerhalb der JVM**

  Um die genannten Nachteile zu vermeiden, wurde im Rahmen dieser Arbeit die folgende Methode entwickelt: Die Messung wird von einer Bibliothek, die in einer anderen Programmiersprache realisiert wurde, durchgeführt. Über eine Brücke zwischen Betriebssystem und JVM, dem *Java Native Interface* (JNI, vgl. Abbildung 5.10), wird der Zugriff von Java auf die Bibliothek ermöglicht. Diese Bibliothek muss allerdings für jede gewünschte Zielplattform programmiert werden. Dafür ist sie sehr kurz (sowohl die MacOS-Version als auch die Linux-Version besteht aus 20 Zeilen C-Code) und erzeugt extrem geringen *Overhead*: Auf einem PC (1,2 GHz Pentium M, 512 MB Hauptspeicher, Linux Kernel 2.6, GCC 3.3) war beim Betrieb von zehn Peers in der Version mit Messung der Prozessorlast keine merkbar höhere Auslastung des Systems festzustellen als in der Version ohne diese Messung.

Aufgrund der Nachteile, die dem Ziel der Lastbalancierung entgegenstehen, wird die erste Methode zur Messung der Prozessorlast nicht weiter betrachtet und ausschließlich die zweite implementiert. Die native Bibliothek enthält eine Funktion, die die Durchschnittslast der letzten *seconds* Sekunden zurück liefert. Diese ruft die Java-Klasse *CPULoadObserver* in der Methode *getLoad* auf (vgl. Abbildung 5.11). Durch die Funktionalität der Basisklasse *Observer* wird mittels der Registrierung eine Balancierung angestoßen, wenn die Lastschwelle überschritten wurde.

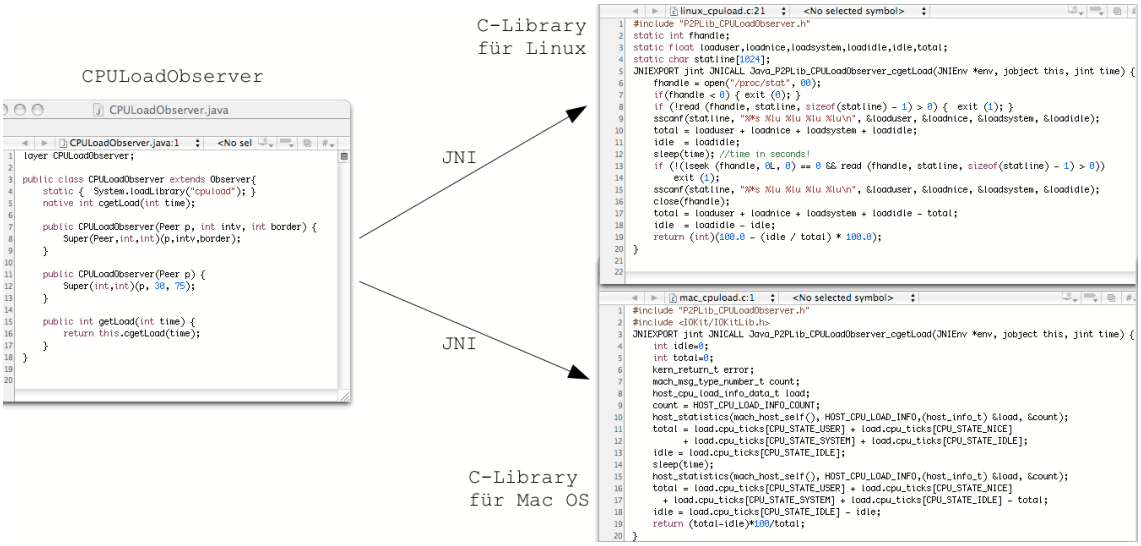

Abbildung 5.11: CPU-Lastmessungscode

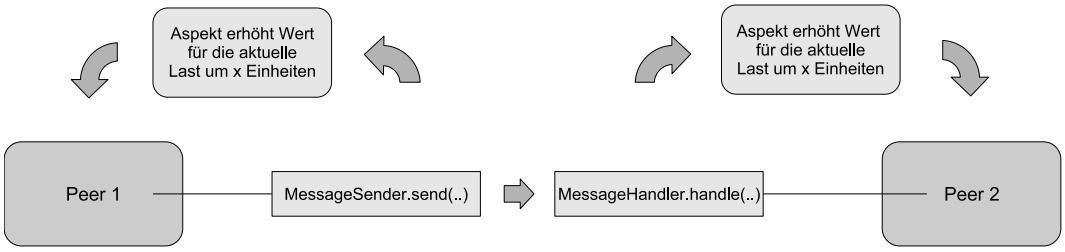

Abbildung 5.12: Ein Aspekt ermittelt die Last eines Peers durch die Anzahl der versendeten beziehungsweise empfangenen Nachrichten

5.3.2 Messung der Anzahl ein- beziehungsweise ausgehender Nachrichten

Bei der Messung der Anzahl der Nachrichten gilt es, zwei Fragen zu beantworten:

1. **Welche Nachrichten sollen gezählt werden?**

   Es können entweder die eingehenden, die ausgehenden oder beide Arten von Nachrichten gezählt werden. Dabei fließen Anfragen an den eigenen Peer nur in die Berechnung ein, wenn die eingehenden Nachrichten berücksichtigt werden. Analog werden eigene Anfragen nur bei Berücksichtigung der ausgehenden Nachrichten mitgezählt. Werden jedoch beide Arten von Nachrichten gezählt, so werden Weiterleitungen doppelt gewichtet. Die im Rahmen dieser Arbeit erstellte Implementation zählt sowohl ein- als auch ausgehende Nachrichten, erkennt jedoch weitergeleitete Nachrichten und gewichtet diese nur einfach. Hierbei werden ausgehende Nachrichten auf ihren Sender überprüft. Handelt es sich nicht um den eigenen Peer, so wird diese Nachricht nicht erneut gezählt.

```
package P2PLib;
public aspect MessageLoadObserverAspect{
    public static int messages=0;
        |
    before(TypedMessageHandler mhand, Message m) : target(mhand)
        && call(* *Handler.handle(Message)) && args(m) {
            messages+=10;
    }
    after(MessageSender msend, Message m, PeerId p) : target(msend)
        && call(* MessageSender.send(Message, PeerId)) && args(m,p) {
            if(msend.getPeer().getId() != null &&
                    msend.getPeer().getId().equals(m.getSource()))
                messages+=10;
    }
}
```

```
layer MessageLoadObserver;
public class MessageLoadObserver extends Observer {
    public MessageLoadObserver(Peer peer) {
        Super(Peer, int, int)(peer, 5, 20);
    }

    public MessageLoadObserver(Peer peer, int time, int border) {
        Super(Peer, int, int)(peer, time, border);
    }

    public int getLoad(int time) {
        MessageLoadObserverAspect.messages=0;
        try {
            Thread.sleep(time*1000);
        } catch(InterruptedException i) {
            interrupt();
        }
        return MessageLoadObserverAspect.messages;
    }
}
```

Abbildung 5.13: Quellcode für die Lastmessung durch Nachrichtenzählung

2. **Wie werden die unterschiedlichen Nachrichten gewichtet?**

   Verschiedene Nachrichtentypen (zum Beispiel Anfrage-, Einfüge- oder Löschnachrichten) können unterschiedlich hohe Last bei einem Peer erzeugen. So sind Einfügeoperationen im allgemeinen besonders aufwendig, da der verhältnismäßig langsame Massenspeicher benutzt wird. Daher werden für die einzelnen Nachrichtentypen Parameter definiert, die den Gewichtungsfaktor repräsentieren. Standardmäßig wird in dieser Arbeit der Gewichtungsfaktor 10 benutzt. So besteht für spezielle Nachrichten ausreichend Spielraum nach oben und unten. Allerdings wird von der Möglichkeit, die Nachrichtentypen unterschiedlich zu gewichten, in dieser Arbeit kein Gebrauch gemacht, da keine Untersuchungen über die korrekten Verhältnisse durchgeführt wurden.

Die Implementierung selbst gestaltet sich durch AOP relativ einfach. Der Aspekt *MessageLoadAspect* (siehe Abbildung 5.13 links) fügt jedem Peer einen Nachrichtenzähler hinzu. Durch einen *Pointcut* beim Versenden (Methode *MessageSender.send()*) beziehungsweise Empfangen (Methode *MessageHandler.handle()*) von Nachrichten wird der Zähler entsprechend dem Gewichtungsfaktor – hier immer 10 – des Nachrichtentyps erhöht (vgl. Abbildung 5.12). Durch ein definiertes Zeitintervall wird in der Methode *MessageLoadObserver.getLoad()* (siehe Abbildung 5.13 rechts) festgelegt, wann der Zähler zurückgestellt wird.

5.4 Implementierung konkreter Methoden zur Lastbalancierung

Analog zur Implementierung der Lastmessungen werden in diesem Abschnitt die ausgewählten Methoden zur Lastbalancierung – Replikation sowie Zonentransfer – beschrieben. Dabei werden insbesondere die notwendigen Modifikationen in der P2P-Familie erläutert. Im Gegensatz zu den parallel zu den Peers laufenden Lastmessungen sind hier tiefgreifende Veränderungen notwendig.

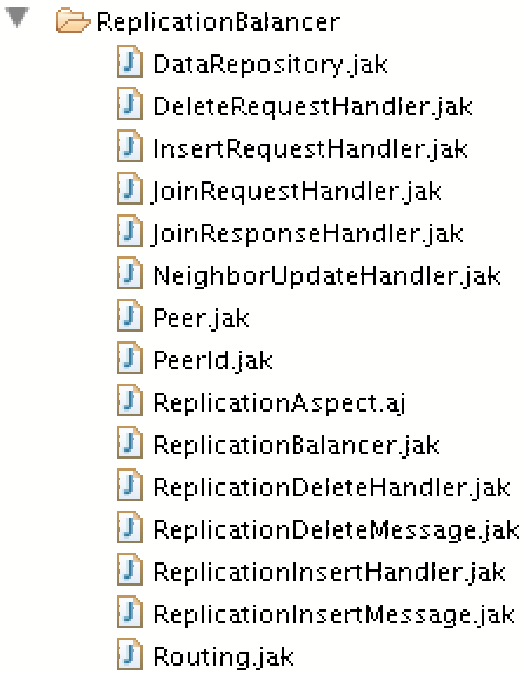

Abbildung 5.14: Übersicht der für die Replikation erstellten *Mixins*/Aspekte

5.4.1 Replikation

Abbildung 5.14 zeigt, dass zur Implementierung des Replikationsmechanismus recht viele *Mixins* (15) notwendig sind. Sie lässt sich in vier Schritte aufteilen:

- Zunächst wird jedem Peer analog zur Verwaltung der Nachbarpeers eine Liste mit Replikaten hinzugefügt. Eine boolesche Variable führt den Zustand des *ReplicationManagers* ein. Nur dieser *ReplicationManager* – es gibt jeweils genau einen pro verwalteter Zone – fordert im Falle einer Hochlast zusätzliche Replikate an. Diese Position nimmt immer der Peer mit der geringsten *PeerId* aller Replikate eines Datenbereichs ein. Dazu wird die entsprechende *compareTo()*-Methode für die Klasse *PeerId* implementiert. Mit dieser Lösung wird verhindert, dass bei Hochlast in einer Zone jeder Peer neue Replikate anfordert. Die Lastbalancierung wäre in diesem Fall zu grobkörnig, da sich so die Anzahl der Replikate in einer Hochlastzone jedes Mal verdoppeln würde.

- Der Aufbau von Replikaten wird durch eine Veränderung im *Join*-Vorgang erreicht. Wie bisher sendet ein neuer Peer bei seinem Start eine Nachricht *(JoinRequestMessage)* an einen Peer, der bereits zum CAN gehört, um diesem CAN beizutreten. Anstatt aber wie bisher seine Zone aufzuteilen und eine Hälfte davon von dem neuen Peer verwalten zu lassen, sendet der bereits im CAN befindliche Peer in der Antwortnachricht *(JoinResponseMessage)* den kompletten eigenen Datenbereich inklusive Nachbarn zurück. Danach fügt er den neuen Peer zu seinen Replika-

ten hinzu. Der neue Peer errechnet sich daraufhin aus den übergebenen Nachbarn seine Replikate heraus. Dabei genügt es, den eigenen minimal verwalteten Schlüssel mit dem jedes Nachbarn zu vergleichen. Nur bei Replikaten sind diese gleich. Danach werden noch – ebenfalls wie bisher – alle Nachbarn mit einer Nachricht *(NeighborUpdateMessage)* über den neuen Peer informiert. Abschließend überprüfen sowohl der neue Peer als auch dessen Nachbarn, ob die maximale Anzahl an erlaubten Replikaten in einer Zone erreicht ist. In diesem Fall teilt sich diese Zone und jeweils die untere und die obere Hälfte der Replikate – ermittelt durch die *compareTo()*-Methode der *PeerIds* – verwaltet nur noch die untere beziehungsweise obere Hälfte des Schlüsselbereichs. Die neue Methode *dropHalf()* der Klasse *DataRepository* übernimmt diese Aufteilung. Auch die Nachbarn einer zu teilenden Zone berechnen die Teilung eigenständig, um den Kommunikationsbedarf zwischen den Peers zu minimieren.

- Die eigentliche Verteilung der Last erfolgt durch die von der Lastmessung aufgerufene Klasse *ReplicationBalancer*, sowie durch das veränderte *Routing*. Wie in Abschnitt 5.2.2 beschrieben, ruft die aktive Lastmessung durch die *LoadRegistry* die Methode *balance()* der Balancierungsklasse, in diesem Fall *ReplicationBalancer*, auf. Falls die maximale Anzahl an Replikaten noch nicht erreicht ist, wird der Nachbarpeer bestimmt, der die geringste Last aufweist und sich in einer Zone mit mehr als der minimalen Anzahl an Replikaten befindet. Diesem Nachbarpeer wird nun eine *JoinResponseMessage* gesendet, so dass er als zusätzliches Replikat zur Entlastung beitragen kann. Durch die Aktualisierung der Nachbarn beziehungsweise der bisherigen Replikate im *JoinResponseHandler* wird der Peer sofort als Replikat in einem neuen Datenbereich registriert.

  Ist die maximale Anzahl der Replikate in einer Hochlastzone erreicht, so wird diese nicht aufgeteilt (wie es beim Anmelden neuer Peers der Fall ist). Bei der Anwendung der P2P-Familie mit dem Replikationsmechanismus stellte sich schnell heraus, dass sonst innerhalb kurzer Zeit eine zu starke Aufteilung stattfindet, in der fast jede Zone nur von der minimalen Anzahl an Replikaten verwaltet wird, so dass keine weiteren Handlungen möglich sind.

  Die Verteilung der Anfragen wird durch ein verändertes *Routing* sichergestellt. Dazu wählt ein Zufallsgenerator zwischen allen Replikaten der besten Nachbarzone gleich verteilt denjenigen Peer aus, dem die Nachricht gesendet wird. Andere Verteilungsalgorithmen lassen sich mit geringem Aufwand ebenfalls in der Klasse *Routing* implementieren.

- Zuletzt müssen noch Einfüge- und Löschoperationen innerhalb der betroffenen Replikate verteilt werden. Dazu verschickt ein Empfänger einer *InsertMessage* beziehungsweise einer *DeleteMessage* diese Nachricht sofort an seine Replikate weiter. Damit die Replikate diese Nachrichten nicht sofort wieder an die anderen Replikate verteilen, werden sie in Form einer *ReplicationInsertMessage* beziehungsweise einer *ReplicationDeleteMessage* versendet. Deren *Handler* verarbeiten sie auf die-

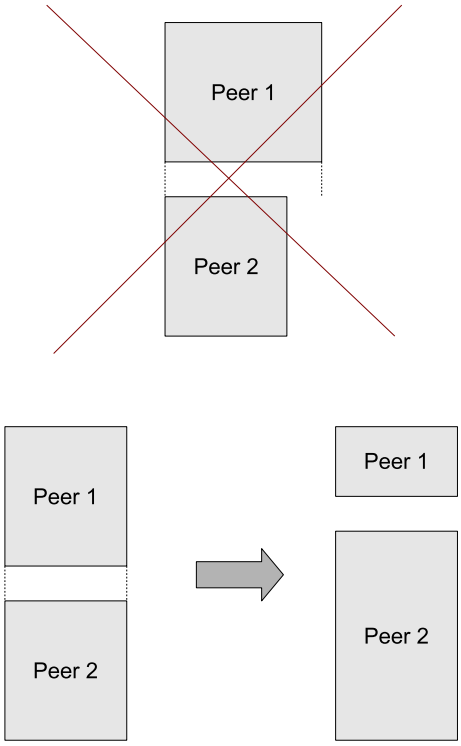

Abbildung 5.15: Prüfung der Übergabemöglichkeit bei der Zonenreorganisation

selbe Weise wie die *InsertMessages* und *DeleteMessages*, allerdings ohne sie weiter zu verteilen.

5.4.2 Zonenreorganisation

Bei der Zonenreorganisation gilt es, einen Teil des von einem Hochlastpeer verwalteten Datenbereichs an einen anderen zu übergeben. Dazu wird zunächst der am geringsten belastete, aber dennoch geeignete Nachbar ausgewählt. Da die Zonen der Peers im CAN nur eine geometrische Form (je nach Dimension zum Beispiel Rechtecke in zweidimensionalen, Quader in dreidimensionalen CANs) annehmen können und diese immer durch zwei Schlüssel – den minimalen und den maximalen Punkt – aufgespannt wird, sind nur bestimmte Nachbarn eines Peers geeignet, um Teile von dessen Zone aufzunehmen. Ein Nachbarpeer ist genau dann geeignet, wenn ausreichend Eckpunkte direkt angrenzen. Für n-dimensionale Schlüsselräume müssen 2^{n-1} Eckpunkte des benachbarten Schlüsselraums an die des Hochlastpeers angrenzen. Zwei Punkte sind angrenzend, wenn ihre Distanz (bestimmt durch die entsprechende Funktion in der Klasse *Key*) genau eine Einheit beträgt. Abbildung 5.15 zeigt die Problematik (zum besseren Verständnis im zweidimensionalen Raum). Bei den oberen beiden Peers ist die Übergabe eines Teils der verwalteten Zone nicht möglich, nur jeweils ein Eckpunkt eines Peers grenzt an den des anderen an. Im unteren Bereich der Abbildung gibt es jeweils zwei angrenzende Eck-

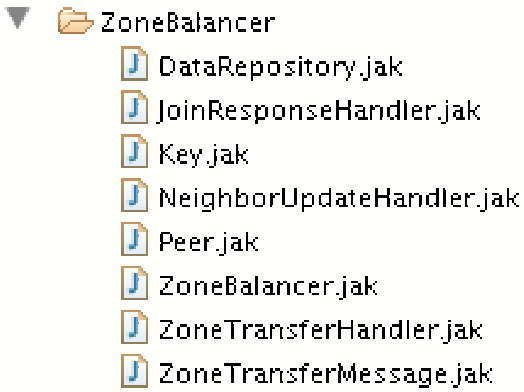

Abbildung 5.16: Übersicht der für die Zonenreorganisation erstellten *Mixins*/Aspekte

punkte, so dass eine Übergabe möglich ist. Ein Parameter bestimmt, wie viel Prozent des eigenen Datenbereichs abgegeben werden. Allerdings hilft ein Parameter von 50% dabei, um die daraus folgenden Zonen möglichst regelmäßig zu halten. Je unregelmäßiger die geometrischen Formen der Zonen sind, desto seltener kann eine Zonenreorganisation überhaupt stattfinden. Im ungünstigsten Fall kann das P2P-Netz in einen Zustand kommen, in dem diese Form der Lastbalancierung überhaupt nicht mehr angewendet werden kann.

In Abbildung 5.16 ist zu erkennen, dass bei der Implementierung der Zonenreorganisation nur etwa halb so viele *Mixins* wie bei der Replikation nötig sind. Bei Hochlast wird *balance()* vom *ZoneBalancer* aufgerufen. Dort wird durch die neue Methode *canSplit()* in der Klasse *DataRepository* überprüft, ob ein Bereich des eigenen Schlüsselraums an einen Nachbarn übergeben werden kann. *getEdges()* der Klasse *Key* berechnet dazu zunächst alle Eckpunkte jeweils durch die gegebenen Schlüssel für den minimalen und den maximalen Bereich. Ergibt sich dabei, dass ein Teil des Schlüsselbereichs übergeben werden kann, so wird die neue eigene (verkleinerte) und die neue (vergrößerte) Nachbar-Zone berechnet und zurückgeben. Der übergebene Bereich wird in einer *ZoneTransferMessage* an den Zielpeer gesendet. Der Empfänger fügt die übergebene Zone anschließend im *ZoneTransferHandler* an seine bisherige an.

5.5 Zusammenfassung

In diesem Kapitel wurde die Implementierung der Lastbalancierung beschrieben. Zunächst musste eine möglichst effiziente Möglichkeit gefunden werden, um die Lastdaten von jedem Peer an dessen Nachbarn zu verteilen. Eine erweiterbare und konfigurierbare Architektur auf Basis der bestehenden P2P-Familie (beschrieben in Kapitel 3) hilft dabei, Redundanz im Code und im Programmieraufwand bei der Implementierung der Lastbalancierung zu verhindern. Abschließend wurden in den Abschnitten über die Implementierung der Lastmessungen und der Lastbalancierungsmethoden verschiedene

Entwurfsmöglichkeiten, die zu konfigurierenden Variablen, sowie die entstandenen *Layer* und die dazugehörigen *Mixins* vorgestellt.

Kapitel 6

Evaluierung

In diesem Kapitel werden die implementierten Methoden zur Lastbalancierung und der Einsatz von AOP beziehungsweise FOP im Bereich Lastbalancierung in einer P2P-Familie evaluiert. Dazu wird zunächst die Experimentierumgebung, sowie ihr grundsätzlicher Aufbau und ihr prinzipieller Einsatz beschrieben. Danach werden Experimente auf Basis realistischer Daten erarbeitet. Nach Durchführung dieser Experimente mit verschiedenen Kombinationen der implementierten Methoden sowie deren Parameter erfolgt eine grafische Aufbereitung der Ergebnisse. Dazu gehört auch eine Diskussion über die Wirksamkeit der verschiedenen Lastbalancierungsmethoden. Abschließend wird der Einsatz von AOP und FOP im Umfeld Lastbalancierung in einer P2P-Programmfamilie bewertet. Dabei werden unterschiedliche Vorgehensweisen und Erleichterungen gegenüber „konventioneller“ Programmierung und Schwierigkeiten bei der Implementierung betrachtet.

6.1 Aufbau und Umgebung der Experimente

Buchmann beschreibt in [BB04b] Web-Skalierbarkeit als Ziel für verteilte Hashtabellen. Dazu müssen realistische Experimente mit einer möglichst großen Anzahl Peers in einem P2P-System durchgeführt werden. Dafür steht dem Institut für Technische und Betriebliche Informationssysteme (ITI) der Universität Magdeburg ein *Cluster* aus 32 Rechnern zur Verfügung. Die Dateien des P2P-Netzes müssen nur auf einem dieser PCs existieren. Von dort werden sie automatisch verteilt. Auch der Aufbau des P2P-Netzes erfolgt zentral. Mit einem Skript erfolgt der Start mehrerer tausend Peers auf dem *Cluster* innerhalb kurzer Zeit. Die Ausgaben aller Peers lassen sich ebenfalls zentral protokollieren. Abbildung 6.1 zeigt das Prinzip dieser Experimentierumgebung. Wegen technischer Schwierigkeiten konnte das *Cluster* allerdings nur für Experimente in der Standard-Konfiguration eingesetzt werden. Bei den Experimenten mit aktivierter Lastbalancierung wurden die Peers nacheinander auf einem Rechner – bei den Versuchen mit Messung der Prozessorlast auf drei Rechnern – gestartet. Aus Mangel an Zeit konnte die Experimentierumgebung nicht an das veränderte Startverhalten der Peers mit

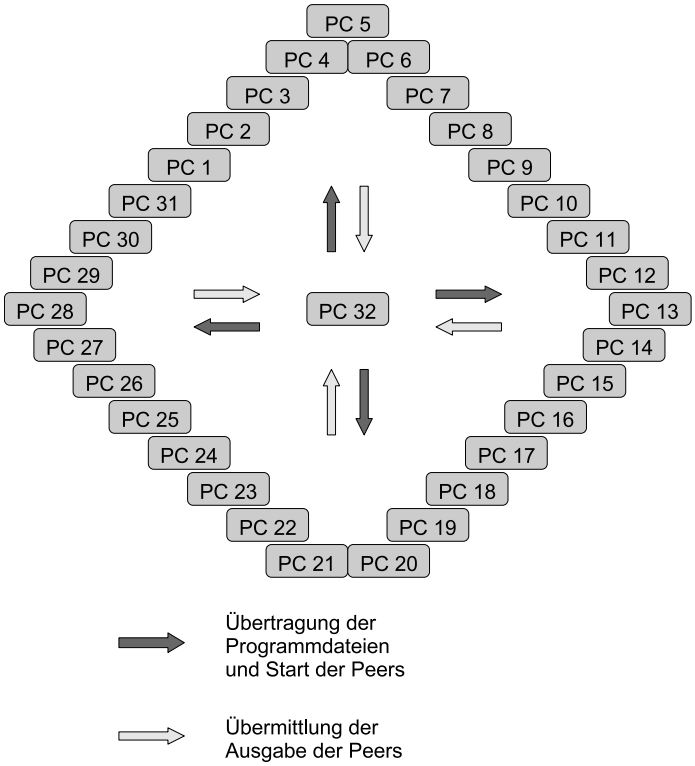

Abbildung 6.1: PC-Cluster aus 32 Rechnern

Lastbalancierung angepasst werden.

6.2 Entwicklung geeigneter Experimente

Im Rahmen dieser Arbeit werden Experimente in drei verschiedenen Kategorien durchgeführt:

1. **Experimente unter Idealbedingungen**

   Als erstes soll die prinzipielle Wirksamkeit der verwendeten Algorithmen zur Lastbalancierung gezeigt werden. Dazu werden Szenarien erstellt, die vermutlich in hohem Maße auf ihre erwarteten Stärken reagieren. Sollte sich hier keine signifikante Verbesserung zeigen lassen, so machen weitere, auf realen Daten basierende Experimente keinen Sinn.

   Die Wirksamkeit der Replikation wird an einem Szenario gezeigt, in dem ausschließlich Anfragen *(Queries)* an das P2P-Netz gestellt werden. Dafür werden drei Konfigurationen der P2P-Programmfamilie verglichen: Einmal ohne jede Lastbalancierung und zwei Konfigurationen mit Replikation. Dort wird einmal die Messung der Prozessorauslastung und einmal die Messung der ein- und ausgehenden Nachrichten für die Lastermittlung benutzt. Geeignete Parameter bei aktivierter

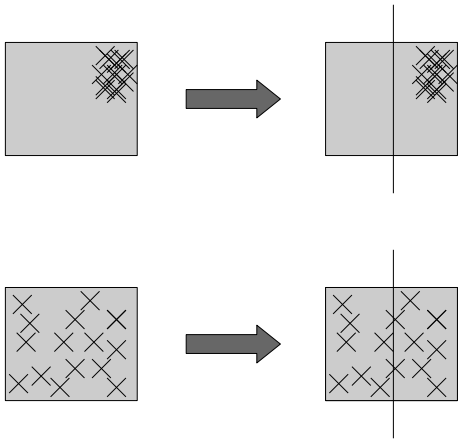

Abbildung 6.2: Gute und schlechte Anfrageverteilung bei der Zonenreorganisation

```
>> 0 (OID=1) 74.4886 7.9906 15.7492 47.6469 10.6278 40.4127 -7.4488 10.5633 5.6775
>> 1 (OID=2) 80.3697 1.8044 5.9979 266.8016 3.8768 28.5198 0.0085 2.0168 68.9624
>> 2 (OID=3) 63.0868 -3.7823 -7.9378 453.8666 19.6060 28.0336 -12.3920 11.6208 -8.7057
>> 3 (OID=4) 70.0130 3.3414 6.7016 309.1751 27.9302 117.5003 18.7777 38.0535 107.4537
>> 4 (OID=5) 73.0952 -5.3458 8.3032 223.7135 63.4224 97.5318 -3.1274 -30.7137 40.7760
>> 5 (OID=6) 75.9065 2.9953 6.8895 260.2926 24.7197 79.1246 -13.1185 22.3480 41.5624
>> 6 (OID=7) 71.7367 7.2272 12.1822 275.4779 25.4680 76.9135 17.2005 17.4782 95.2768
>> 7 (OID=8) 68.2078 1.5344 -4.5546 164.3120 12.0735 60.8609 23.7818 20.8300 77.1509
>> 8 (OID=9) 60.5823 -8.7616 -7.7828 282.3165 100.0918 63.9244 -34.3561 -20.0019 -10.5520
>> 9 (OID=10) 84.5085 -3.6213 -4.1088 37.7763 14.2863 61.2462 13.3513 26.5481 25.0092
>> 10 (OID=11) 54.1388 -12.0706 -35.6339 40.2522 73.3768 35.2777 7.4044 29.6094 29.0087
>> 11 (OID=12) 74.3051 3.6233 5.3340 352.1357 12.5549 49.1101 27.7647 18.8608 91.4553
>> 12 (OID=13) 73.4974 -4.4086 -1.2440 356.9255 24.7896 36.5603 50.2026 -15.1748 -7.5043
>> 13 (OID=14) 71.1411 0.5690 -8.0461 28.8910 7.5357 20.1025 9.7291 11.0975 17.8275
>> 14 (OID=15) 72.7445 -2.7058 -0.6370 277.0435 29.0737 34.2969 15.0433 7.1575 13.5026
>> 15 (OID=16) 71.1141 1.9801 2.4737 327.7742 7.3343 27.2746 26.9607 9.8215 49.4547
>> 16 (OID=17) 66.3928 17.4299 26.8877 218.1318 278.7742 470.3773 35.6619 265.3186 158.1621
```

Abbildung 6.3: Ausschnitt der Merkmalsvektoren

Lastbalancierung werden empirisch durch mehrfache Probeläufe mit verschiedenen Werten ermittelt (das gleiche gilt für alle folgenden Experimente).

Ein weiteres Szenario untersucht die Wirksamkeit der Zonenreorganisation. Im Gegensatz zum zuvor beschriebenen Experiment muss hier nicht auf Einfügeoperationen verzichtet werden. Allerdings müssen die Operationen möglichst gut innerhalb einer (Anfangs-)Zone verteilt sein, da nur so mit der Übergabe von Teilen der Zone an einen Nachbarpeer auch ein Teil der Last übertragen wird. Abbildung 6.2 verdeutlicht diesen Sachverhalt. Im oberen Teil sind die Anfragen schlecht innerhalb einer Zone verteilt. Eine Aufteilung der Zone (durch die Zonenreorganisation) teilt demzufolge die Last nicht zwischen zwei Peers auf. Anders im unteren Teil der Abbildung: Durch die ausgewogene Verteilung der Anfragen wird die Last nach der Zonenreorganisation ebenfalls verteilt.

2. **Experimente mit realen Bilddaten**

   Für diese Arbeit standen Daten von etwa 300.000 Bilder aus dem Internet zur

```
195.93.102.69 - - [19/Feb/2005:14:28:42 +0100] "GET
195.93.102.69 - - [19/Feb/2005:14:28:42 +0100] "GET
195.93.102.38 - - [19/Feb/2005:14:28:42 +0100] "GET
195.93.102.37 - - [19/Feb/2005:14:28:42 +0100] "GET
195.93.102.37 - - [19/Feb/2005:14:28:42 +0100] "GET
195.93.102.38 - - [19/Feb/2005:14:28:43 +0100] "GET
195.93.102.42 - - [19/Feb/2005:14:28:43 +0100] "GET
195.93.102.38 - - [19/Feb/2005:14:28:43 +0100] "GET
195.93.102.66 - - [19/Feb/2005:14:28:43 +0100] "GET
195.93.102.6 - - [19/Feb/2005:14:28:43 +0100] "GET
195.93.102.67 - - [19/Feb/2005:14:28:43 +0100] "GET
195.93.102.37 - - [19/Feb/2005:14:28:44 +0100] "GET
195.93.102.37 - - [19/Feb/2005:14:28:44 +0100] "GET
195.93.102.69 - - [19/Feb/2005:14:28:44 +0100] "GET
195.93.102.36 - - [19/Feb/2005:14:28:45 +0100] "GET
195.93.102.71 - - [19/Feb/2005:14:28:45 +0100] "GET
195.93.102.36 - - [19/Feb/2005:14:28:45 +0100] "GET
195.93.102.66 - - [19/Feb/2005:14:28:45 +0100] "GET
195.93.102.10 - - [19/Feb/2005:14:28:45 +0100] "GET
195.93.102.67 - - [19/Feb/2005:14:28:46 +0100] "GET
195.93.102.69 - - [19/Feb/2005:14:28:46 +0100] "GET
195.93.102.72 - - [19/Feb/2005:14:28:46 +0100] "GET
```

Abbildung 6.4: Ausschnitt des Zugriffprotokoll vom *Apache-Webserver*

Verfügung. Die Bilder durchliefen diverse Algorithmen zur Merkmalsextraktion. So wurden zahlreiche hochdimensionale Merkmalsvektoren erstellt. Zum Beispiel im Zusammenhang mit dem *Semantic Web*[1], wo neben Informationen auch deren Klassifikation und Annotation gespeichert wird, erscheint eine Speicherung dieser Vektordaten innerhalb einer P2P-Datenstruktur sinnvoll. So kann beispielsweise erreicht werden, dass nur Benutzer auf fremde Klassifikationen zugreifen können, die ihrerseits auch die der eigenen Daten anbieten.

Abbildung 6.3 zeigt einen Original-Ausschnitt mit einigen der Vektoren. Diese werden hier verwendet, um eine realistische Verteilung von Daten innerhalb eines P2P-Netzes zu generieren. Ein Skript berechnet einmalig eine zufällige Abfolge von Einfüge- und Abfrageoperationen an die Punkte mit den Koordinaten der Vektoren. Diese Folge von Operationen wird dann auf allen Konfigurationen durchgeführt. Da hier realistische Experimente durchgeführt werden sollen, müssen diese auch mit einer möglichst großen Anzahl Peers durchgeführt werden. Allerdings hilft die Lastermittlung durch Messung der Prozessorlast nur, wenn auf jedem PC genau ein Peer läuft. Daher wird diese Form der Lastmessung im folgenden nicht mehr verwendet.

3. **Experimente auf Basis von *Webserver*-Protokolldateien**

   Die zweite Serie von realistischen Experimenten basiert auf Protokoll-Dateien von einem *Webserver*. Dazu wird die etwa 435 *Megabyte* große Zugriffsdatei *access_log* eines produktiv im Einsatz befindlichen *Apache-Servers*[2] verwendet. Diese Datei

[1] http://www.w3.org/2001/sw/
[2] http://httpd.apache.org/

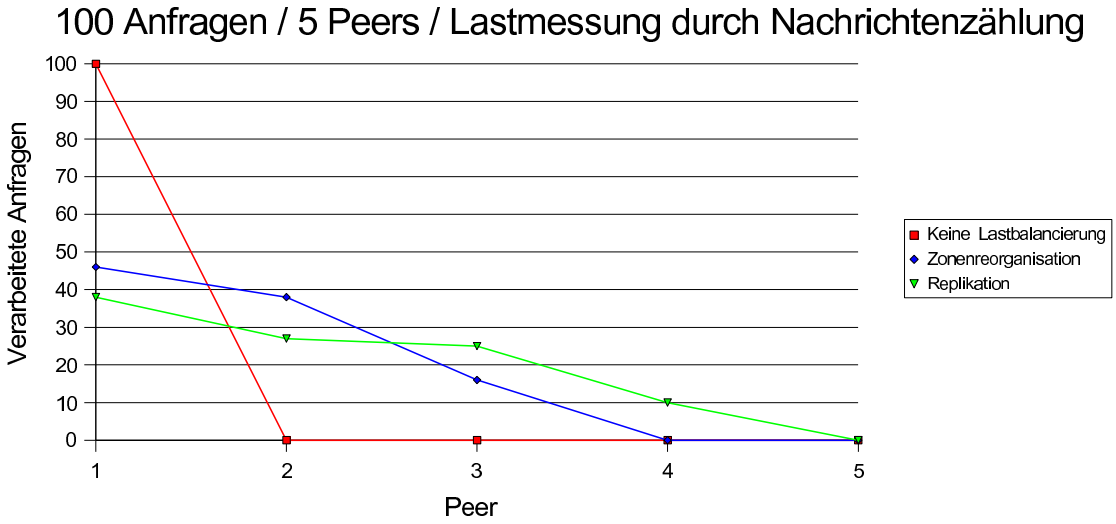

Abbildung 6.5: Experiment 1 unter Idealbedingungen

besteht aus insgesamt 2,15 Millionen Einträgen, die unter anderem die IP-Adressen der auf die Webseiten zugreifenden Rechner enthalten. Diese IP-Adressen werden hier als Punkte im P2P-Netz betrachtet. So ensteht eine realistische Verteilung sowohl der Daten als auch der Zugriffe auf diese Daten. Damit wird zum Beispiel ein Szenario abgebildet, in dem ein verteilter Informationsspeicher vergleichbar mit dem Internet vollkommen dezentral eingesetzt wird (im Gegensatz zum Internet, in dem zum Beispiel zentrale Knotenpunkte existieren). Dazu kann unter anderen auch die zuvor beschriebene Speicherung von Vektoren mit Bildinformationen auf Basis eines IP-basierten Netzwerks gehören.

6.3 Ergebnisse der Experimente

1. **Experimente unter Idealbedingungen**

   Abbildung 6.5 zeigt das erste Ergebnis der Experimente. Dabei wurde die Last mit Hilfe der Anzahl der verarbeiteten Nachrichten ermittelt. Die Lastverteilung durch Zonenreorganisation beziehungsweise Replikation wird einem Versuch ohne Lastverteilung gegenübergestellt. Bei diesem für die Lastverteilungsmechanismen günstigen Zustand ist deutlich die prinzipielle Wirksamkeit zu erkennen. Je gleichmäßiger die Last verteilt ist, desto geringer ist der Abfall in der Kurve (die jeweils die Anzahl der verarbeiteten Nachrichten pro Peer abbildet). Während ohne Lastbalancierung alle 100 Anfragen von einem Peer beantwortet werden, verteilt sich die Bearbeitung bei Zonenreorganisation auf drei, bei Replikation auf vier Peers. Die Varianz der Anfrageverteilung steigt in allen Experimenten, die im Rahmen dieser Arbeit durchgeführt wurden, mit zunehmender Anzahl von Anfragen. Abbildung 6.6 zeigt jedoch deutlich, dass die Varianz ohne Lastbalancierung deutlich

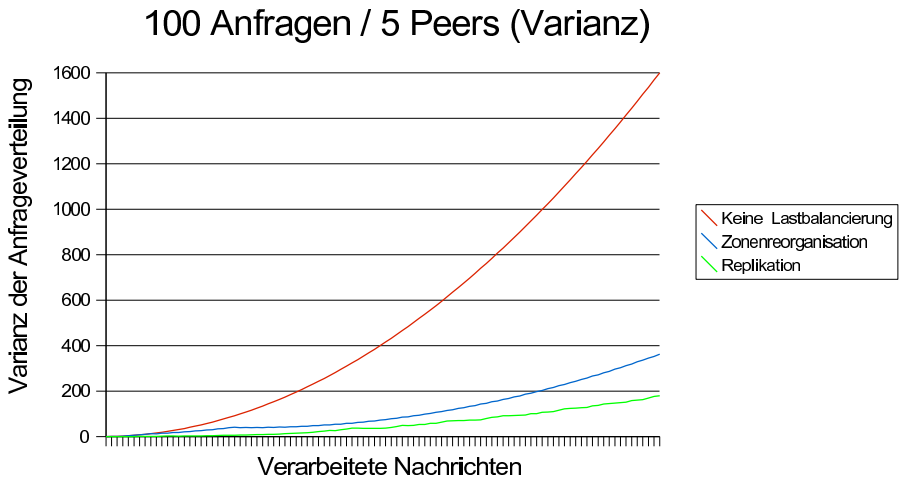

Abbildung 6.6: Verlauf der Varianz bei Experiment 1

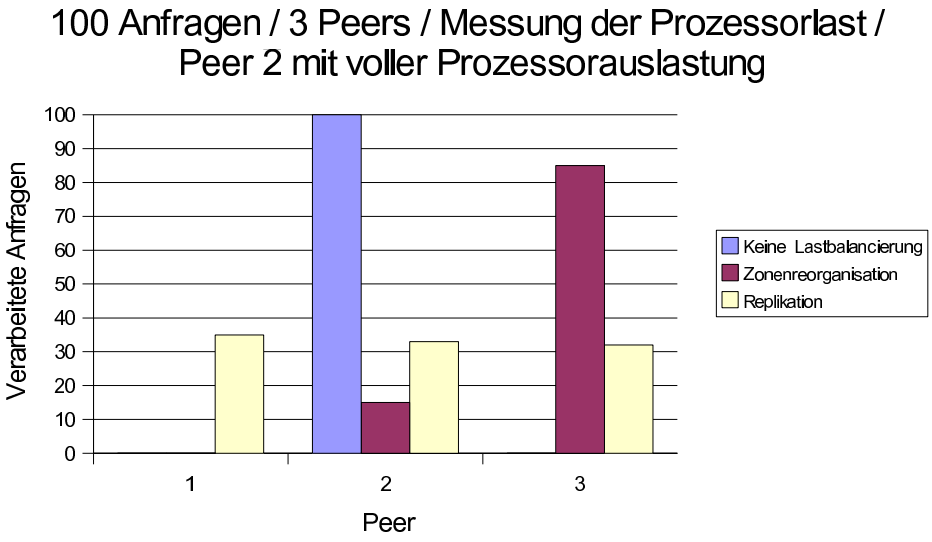

Abbildung 6.7: Experiment 2 unter Idealbedingungen

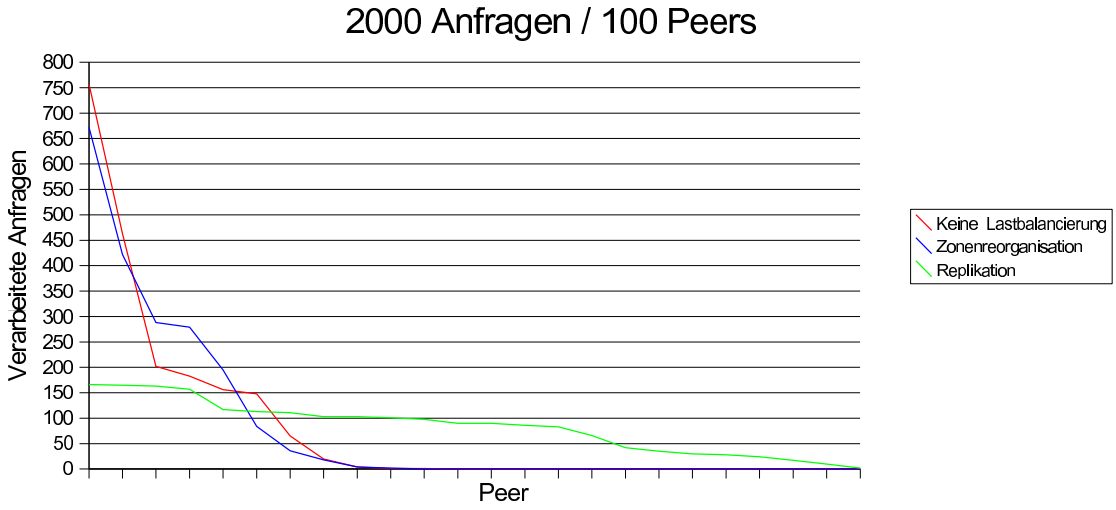

Abbildung 6.8: Experiment mit realen Bilddaten

stärker steigt.

Dass hochbelastete Peers mit Hilfe der Messung der Prozessorlast entlastet werden können zeigt Abbildung 6.7. Der Prozessor des Rechners von Peer 2 wurde dazu durchgängig zu 100% belastet. Es ist deutlich zu erkennen, dass durch die Lastbalancierung die anderen Peers einen Großteil der Anfragen übernehmen. Durch die Replikation wurden alle drei Peers in etwa gleich belastet, durch die Zonenreorganisation konnten etwa 85% der anfallenden Anfragen an einen Niederlastpeer (Peer 3) übertragen werden. Die Replikation könnte den Hochlastpeer dann noch effektiver von Last befreien, wenn die Anfrageverteilung im Replikationsmechanismus die Last der jeweiligen Replikate berücksichtigt. Aktuell wird sie durch einen Zufallsgenerator in etwa gleich verteilt.

2. **Experimente mit realen Bilddaten**

   Abbildung 6.8 und 6.9 visualisieren die Ergebnisse einer Versuchsreihe mit insgesamt 2000 Anfragen auf Basis von Bildvektoren, die an 100 Peers verteilt wurden. Dort ist zu erkennen, dass die Last beim P2P-System im Ausgangszustand schlechter verteilt ist als bei aktivierter Lastbalancierung. Damit wurde das eigentliche Ziel – eine gleichmäßigere Verteilung der Anfragen – erreicht. Die Zonenreorganisation kann jedoch nur eine recht geringe Verbesserung bewirken. Mit dem Replikationsmechanismus werden deutlich bessere Werte erzielt. Dann werden allerdings auch wesentlich mehr Nachrichten im Hintergrund verschickt (eine Einfüge- oder Löschoperation muss an alle Replikate, die eine Zone verwalten, verteilt werden).

3. **Experimente auf Basis von *Webserver*-Protokolldateien**

   Auch bei den Experimenten, deren Anfrageverteilung auf Einträgen in einem Webserver-Protokoll basieren, konnte mit Hilfe der implementierten Lastbalancie-

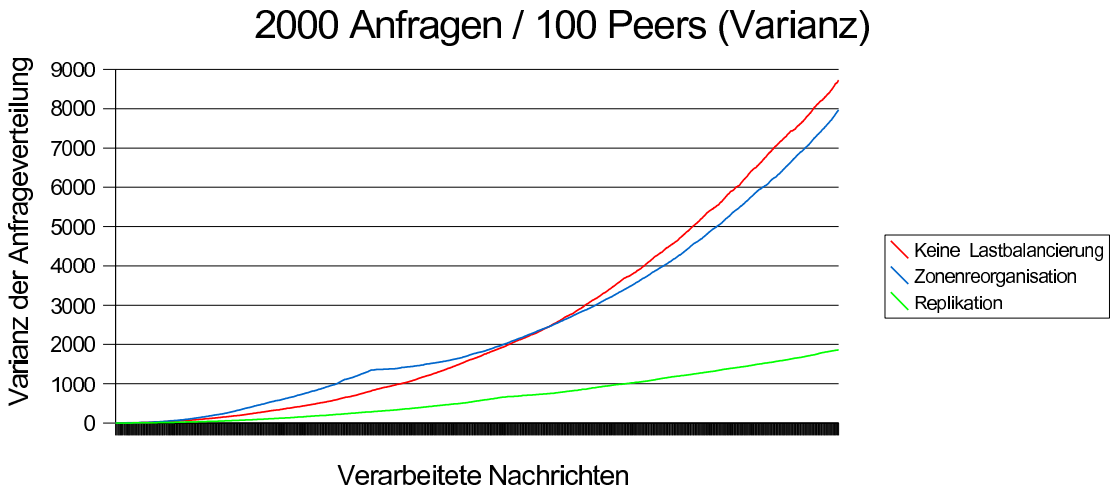

Abbildung 6.9: Verlauf der Varianz bei Anfragen auf Basis von Bildvektoren

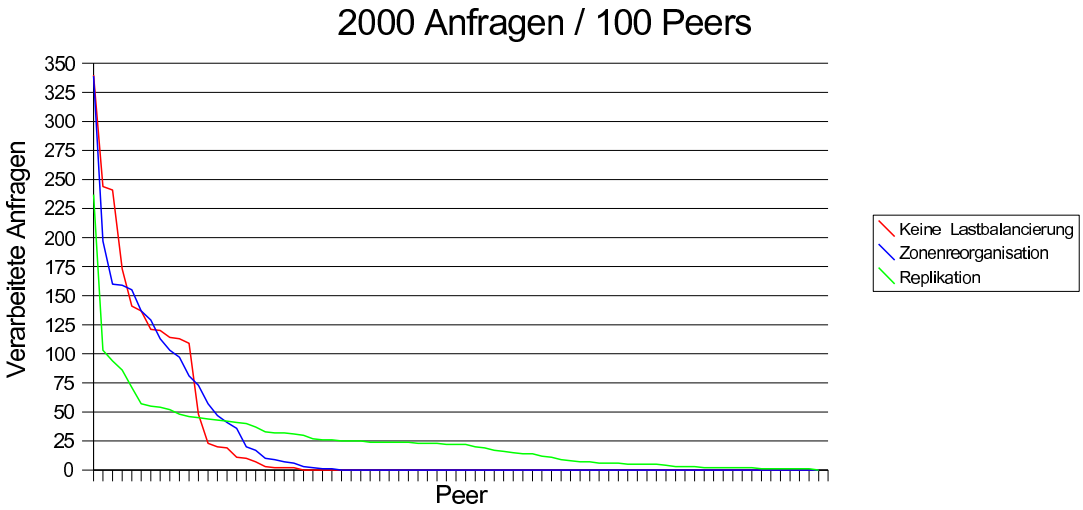

Abbildung 6.10: Experiment mit Daten aus einem Webserver-Protokoll

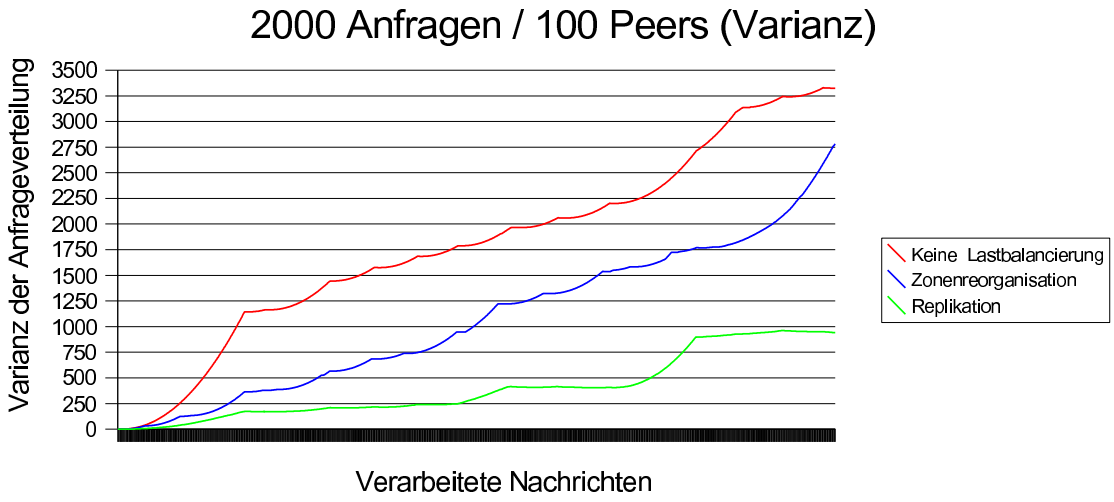

Abbildung 6.11: Verlauf der Varianz bei Anfragen auf Basis eines Webserver-Protokolls

rung eine Verbesserung erreicht werden. Allerdings ist der Effekt der Zonenreorganisation wieder deutlich geringer als bei der Replikation. In diesem Fall lässt sich das damit erklären, das die Protokoll-Einträge oft vielfach hintereinander dieselbe IP-Adresse beinhalten (die hier zur Zielschlüsselgenerierung für die Anfragen verwendet wurden). Die daraus entstehende Situation lässt sich mit der aus Abbildung 6.2 (oberer Teil) vergleichen. Eine Neuaufteilung einer belasteten Zone bringt so keine oder nur geringe Entlastung für den Peer, der sie verwaltet.

Zusammenfassend kann festgestellt werden, dass die implementierten Methoden zur Lastbalancierung in jedem der durchgeführten Experimente eine Verbesserung gegenüber dem Ausgangs-P2P-System erreicht haben. Dabei war der Replikationsmechnismus im allgemeinen erfolgreicher. Dafür wird dort zusätzliche Kommunikation zwischen den Peers durch die Synchronisierung der Replikate generiert. Auch wird insgesamt mehr Speicherplatz benötigt, da jedes Objekt in allen Replikaten der Zone, in der es liegt, existieren muss. Die Lastbalancierung mittels Zonenreorganisation unterliegt diesen Beschränkungen nicht. Allerdings ist sie nur dann besonders hilfreich, wenn die Anfragen gut innerhalb einer Zone verteilt sind (vgl. Abbildung 6.2). Das wichtigste Ergebnis der durchgeführten Experimente besteht also in der Feststellung, dass die Auswahl der besten Methode zur Lastbalancierung in hohem Maße vom gewünschten Einsatzgebiet abhängt. Entwickler von Anwendungen auf Basis der P2P-Familie sollten folglich verstehen, auf welche Weise die verschiedenen Lastbalancierungsmechanismen funktionieren, um die für sie geeignete(n) zu finden. Noch besser wäre natürlich die Implementation eines P2P-Netzes, das sich selbstständig zur Laufzeit an die jeweils aktuellen Anforderungen anpasst.

6.4 Einsatz von AOP und FOP

Der Einsatz aspektorientierter sowie featureorientierter Programmierung erfolgte weitgehend problemlos. Das Ziel, eine komplett separierte und konfigurierbare Lastbalancierung zu implementieren, wurde erreicht. Die Programmierung benötigte in etwa vier Mannwochen, wobei die Implementierung der Lastverteiler den größten Teil der Zeit beanspruchte. Es entstanden insgesamt 30 *Mixins* in sieben *Layer*, drei Aspekte und zwei native Bibliotheken. Die komplette hier realisierte Lastbalancierung besteht aus etwa 1400 Zeilen Quellcode, von denen etwa 1000 die beiden Methoden zur Lastverteilung umsetzen. Daraus lässt sich erkennen, dass das Gerüst der Lastbalancierung und die Implementierung von Methoden zur Lastmessung und Lastdatenverteilung mit relativ wenig Aufwand programmiert werden konnten. Die Lastverteilung muss jedoch an vielen Stellen in das existierende System eingreifen und diese ändern, so dass deren Implementierung recht aufwendig gerät. Insgesamt kann festgestellt werden, dass es sich bei der Lastbalancierung, insbesondere bei der Nutzung verschiedener Methoden, um ein sehr „großes“ *Feature* handelt – die Anzahl der Quellcodezeilen der P2P-Familie erhöhte sich im Rahmen dieser Arbeit um mehr als 15%. Durch die weitere Aufteilung in Teil-*Features* (vgl. Kapitel 5) bleibt die Wartbarkeit jedoch erhalten.

Es stellte sich jedoch im Laufe der Implementierung heraus, dass AOP ausschließlich zum Start der Lastbalancierung, zur einfachen Messung von ein- und ausgehenden Nachrichten, sowie zur passiven Lastdatenverteilung benötigt wurde. In erster Linie wird die aspektorientierte Programmierung also als *nicht-invasive* Verbindung zwischen der ursprünglichen P2P-Familie und den verschiedenen Implementierungen von Lastbalancierung benötigt. Durch geschickten Einsatz von FOP könnte aber sicherlich auch auf diese Aspekte verzichtet werden. So müsste dann zum Beispiel die Nachrichtenzählung durch eine Erweiterung der Nachrichtensender *(MessageSender)* beziehungsweise -empfänger *(MessageHandler)* realisiert werden. Dort würde dann wahrscheinlich auch die passive Lastdatenverteilung umgesetzt. Die Tatsache, dass hier theoretisch auf AOP verzichtet werden kann – durch den Einsatz von FOP sogar ohne den Verlust von Modularität – ist insofern bemerkenswert, da AOP von beiden Technologien technisch gesehen als deutlich leistungsfähiger erscheint. So bietet *AspectJ* unter anderem die Möglichkeit, flexibel alle Methodenaufrufe, die einem bestimmten Muster entsprechen (zum Beispiel deren Namen mit *get* anfängt), zu beobachten oder zu verändern. Da die featureorientierte Programmierung im wesentlichen „nur“ eine Art konfigurierbare Vererbung anbietet, muss für jede Änderung einer Methode die Klasse, in der sie vorkommt, erweitert werden. Die Zusammenhänge zwischen beiden Techniken werden unter anderem in [ALRS05] und [MO04] näher beschrieben. Die ausschließliche Nutzung von FOP hätte zudem den Vorteil, dass sie sich wahrscheinlich noch besser in die (hauptsächlich FOP-basierende) P2P-Programmfamilie eingliedern würde.

Der Einsatz der ATS ermöglicht eine einfache Auswahl der Methoden zur Lastbalancierung. Diese Auswahl muss dann nur noch im Hauptaspekt gestartet werden. Die Aufteilung der P2P-Programmfamilie in einzelne *Features* half in hohem Maße, den (nahezu undokumentierten) Quellcode zu verstehen. Die Einarbeitung in alle Prinzipien des

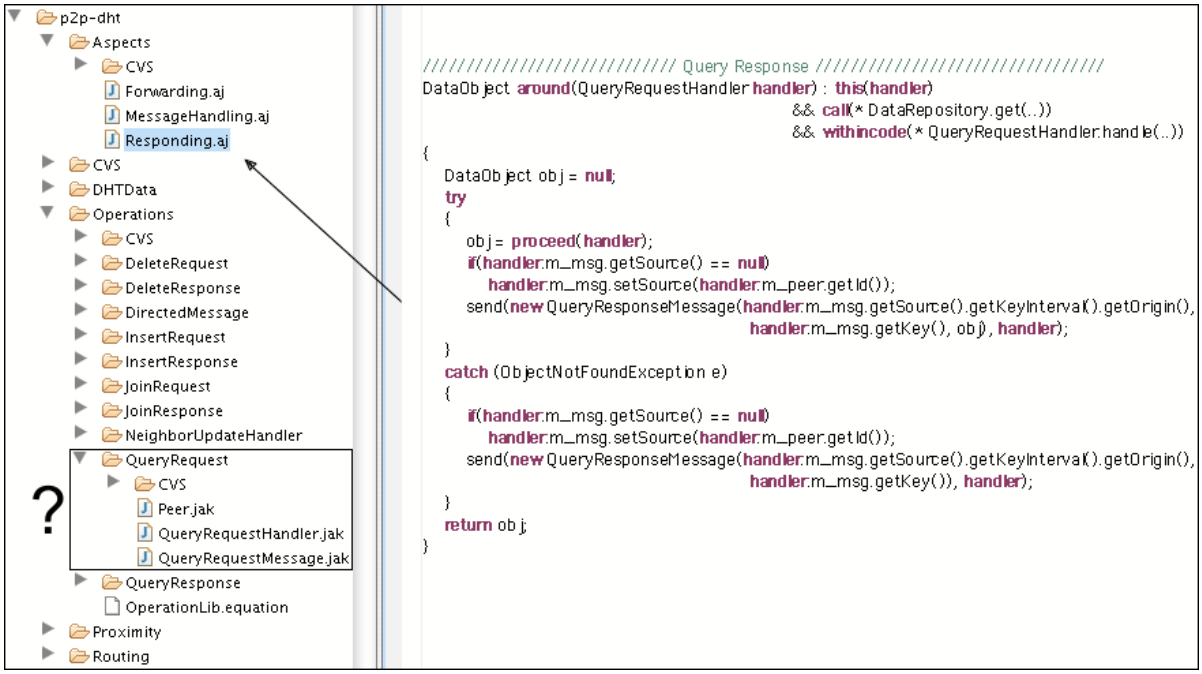

Abbildung 6.12: Probleme bei der Integration von AOP und FOP

Systems hat dennoch mehrere Wochen in Anspruch genommen. Ohne die vorhandene Struktur wäre diese Zeitspanne mit Sicherheit deutlich größer ausgefallen.

Die mangelnde Integration von ATS und AspectJ hat nicht nur leicht erhöhten Aufwand bei der Konfiguration zur Folge (neben der deklarativen Konfigurationsdatei müssen bei den Aspekten noch Veränderungen oder Kommentare hinzugefügt beziehungsweise entfernt werden). Schwerer wiegen Aspekte, die das Verhalten von bestimmten Klassen ändern, ohne dass über *Layer* der inhaltliche Bezug hergestellt wird. Um das zu verdeutlichen, wird in Abbildung 6.12 ein Ausschnitt aus einem Teil der P2P-Programmfamilie (beschrieben in Abschnitt 3.3) gezeigt. Der Aspekt *Responding* sendet bei erfolgreicher Bearbeitung einer Anfrage im *QueryRequestHandler* die Antwort an den anfragenden Peer (siehe Quellcode auf der rechten Seite der Abbildung). Ohne Detailkenntnis der Programmfamilie kann man den Quellcode für diese Aktion leicht im *Layer QueryRequest* (siehe unten links auf der Abbildung) vermuten. Durch solche Aspekte kann die klare Struktur einer Programmfamilie verloren gehen.

Abgesehen von den wenigen hier beschriebenen Problemen haben sich beide Technologien erfolgreich bewährt. Trotz fehlender Unterstützung durch Entwicklungsumgebungen[3] und Abwesenheit eines *Debuggers* zur Fehlersuche gelang die Implementierung ohne große Probleme. Dabei half *AspectJ* vor allem durch die Möglichkeit, Erweiterungen modular und mit wenig Quellcode zu realisieren. Der Einsatz der ATS bringt neben einfacher Konfiguration vor allem eine leicht nachvollziehbare Struktur in die Programmfamilie.

[3]Die Entwicklungsumgebung *Eclipse* unterstützt nur den alleinigen Einsatz von *AspectJ*.

Kapitel 7

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden erfolgreich verschiedene Methoden zur Lastbalancierung modular in einer P2P-Programmfamilie implementiert. Dazu wurden als erstes die zum Verständnis der Arbeit notwendigen Grundlagen über P2P-Systeme und die verwendeten Softwaretechniken – aspektorientierte und featureorientierte Programmierung sowie Programmfamilien – vorgestellt. Danach wurde die Funktionalität der P2P-Programmfamilie und ihre Struktur erläutert. Es folgte eine Einführung in die Lastbalancierung. Zunächst wurden grundlegende Begriffe und Ziele von Lastbalancierung, sowie verschiedene Arten von Last definiert. Anschließend wurden aus der Literatur bekannte Methoden zur Lastbalancierung zusammengefasst. Mit der Implementierung eines Teils dieser Methoden (Replikation und Zonenreorganisation) beschäftigte sich das darauf folgende Kapitel. Dort wurde auch die Entwicklung einer flexibel erweiterbaren und konfigurierbaren Architektur sowie die Implementierung der Lastdatenverteilung und der Lastmessung besprochen. In Kapitel 6 wurden dann einerseits die implementierte Lastbalancierung und andererseits die eingesetzten softwaretechnischen Methoden evaluiert.

Die wichtigsten Ergebnisse dieser Arbeit zusammengefasst sind:

- Die klare Definition grundlegender Begriffe rund um die Lastbalancierung. An Stellen, wo bekannte Literatur andere Definitionen benutzt, wurde die entwickelte Definition begründet. Außerdem werden die Ziele der Lastbalancierung systematisiert und klassifiziert.

- Die Integration einer leicht erweiterbaren Architektur zur Lastbalancierung in die existierende P2P-Programmfamilie. Neue Methoden zur Lastmessung oder zur Lastverteilung können auf sehr einfache Weise hinzugefügt werden.

- Entwurf von Experimenten zur Evaluierung der Effektivität der Lastbalancierung. Neben Experimenten zur prinzipiellen Wirksamkeit wurden vor allem zwei Versuchsreihen auf Basis realer Daten entwickelt. Dabei zeigte sich, dass die implementierten Methoden deutliches Potential zum Erreichen der Ziele der Lastbalan-

cierung zeigen. In den praxisrelevanten Experimenten war der Erfolg allerdings geringer ausgeprägt als dort, wo Idealzustände hergestellt wurden.

- Die Bewertung der Eignung von AOP und FOP im Zusammenhang von Lastbalancierung in einer P2P-Programmfamilie. Mit ihrer Hilfe gelang die Programmierung ohne große Schwierigkeiten. Ein wenig überraschend stellte sich heraus, dass das technisch leistungsfähigere *AspectJ* (als Vertreter von AOP) zur modularen Implementierung nicht zwangsweise benötigt wird und ohne echte Nachteile darauf verzichtet werden könnte. Die mangelnde Integration von ATS und *AspectJ* sorgt für minimal höheren Arbeitsaufwand bei der Konfiguration und für etwas geringere Übersicht bei der Struktur der Programmfamilie.

Im Laufe der Arbeit ergaben sich einige weiterführende Themen, deren Bearbeitung weitere Verbesserungen oder Erkenntnisse im Umfeld von Lastbalancierung in P2P-Systemen ergeben kann:

- Es stellte sich heraus, dass allein mit den bereits implementierten Methoden zur Lastbalancierung eine Vielzahl von variablen Parametern eingeführt werden musste. Dazu gehört beispielsweise die minimale sowie die maximale Anzahl von Replikaten in einer Zone oder der Grad an Auslastung, ab der ein Peer als Hochlastpeer reagiert (und damit Unterstützung anfordert). Allerdings wurden gute Werte dieser Parameter für die Experimente nur sehr grob empirisch ermittelt. Eine weiterführende Arbeit könnte sich ausschließlich mit dem Thema beschäftigen, wie man möglichst optimale Werte zu den jeweiligen Zielszenarien bestimmen kann. Denkbar wäre auch die Entwicklung von Methoden zur dynamischen Anpassung der Parameter zur Laufzeit. Dadurch könnten die bereits entwickelten, praxisrelevanten Experimente noch erfolgreicher verlaufen.

- Ein beliebtes Thema in der aktuellen Forschung ist die in Abschnitt 2.2 kurz vorgestellte Adaption biologischer Algorithmen für IT-Systeme. Einige davon können vielleicht helfen, die Leistungsfähigkeit der Lastbalancierung zu erhöhen. Diese Arbeit bildet für deren Implementierung eine wichtige Grundlage, da sowohl ein Mechanismus zur Metadatenverteilung als auch eine flexible Architektur entwickelt wurden. Darauf aufbauend können verschiedene biologische Algorithmen (zum Beispiel die in dieser Arbeit kurz vorgestellten) implementiert und unter anderem anhand der ebenfalls hier entwickelten Experimente vergleichend auf Wirksamkeit untersucht werden.

- Ein weiteres Forschungsfeld im Rahmen von P2P-Systemen sind Reputationsmechanismen. Die im Rahmen dieser Arbeit entstandene Lösung berücksichtigt in keiner Weise, dass es auch unkooperative Peers geben kann. Im Gegenteil, es wird aus Effizienzgründen voll darauf vertraut, dass alle Peers immer korrekte Informationen liefern. Ein Teilnehmer eines P2P-Systems könnte das ausnutzen und die eigene Arbeit so gering wie möglich halten, indem er seinen Nachbarn immer

mitteilt, dass er voll ausgelastet ist. Die Kombination von Mechnismen zur Lastbalancierung und zur Reputationsermittlung ist daher ebenfalls ein offenes Thema.

Literaturverzeichnis

[AB05] Apel, S.; Böhm, K.: Self-Organization in Overlay Networks. In *Proceedings of the 1st International CAISE Workshop on Adaptive and Self-Managing Enterprise Applications, Porto, Portugal*, Juni 2005.

[ABCM04] Andrade, N.; Brasileiro, F. V.; Cirne, W.; Mowbray, M.: Discouraging Free Riding in a Peer-to-Peer CPU-Sharing Grid. In *13th International Symposium on High-Performance Distributed Computing, Honolulu, Hawaii, USA*, Juni 2004.

[ALRS05] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering, Tallinn, Estonia*, September 2005.

[ATS04] Androutsellis-Theotokis, S.; Spinellis, D.: A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, Band 36, Nr. 4, Dezember 2004.

[BA05] Buchmann, E.; Apel, S.: Piggyback Meta-Data Propagation in Distributed Hash Tables. In *Proceedings of the International Conference on Web Information Systems and Technologies, Miami, Florida*, Mai 2005.

[BB03] Buchmann, E.; Böhm, K.: Effizientes Routing in verteilten skalierbaren Datenstrukturen. In *Proceedings der 10. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web, Leipzig, Germany*, Februar 2003.

[BB04a] Buchmann, E.; Böhm, K.: FairNet - How to Counter Free Riding in Peer-to-Peer Data Structures. In *Proceedings of the International Conference on Cooperative Information Systems, Agia Napa, Cyprus*, Oktober 2004.

[BB04b] Buchmann, E.; Böhm, K.: How to Run Experiments with Large Peer-to-Peer Data Structures. In *Proceedings of the 18th Int. Parallel and Distributed Processing Symposium, Santa Fe, USA*, April 2004.

[BCM03] Byers, J.; Considine, J.; Mitzenmacher, M.: Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the 2nd International Workshop on Peer-To-Peer Systems, Berkeley, California, USA*, Februar 2003.

[BHV01] Binder, W.; Hulaas, J. G.; Villazón, A.: Portable Resource Control in Java. In *Proceedings of the 16th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tampa Bay, Florida, USA*, Oktober 2001.

[BKM05] Bienkowski, M.; Korzeniowski, M.; Meyer, F.: Dynamic Load Balancing in Distributed Hash Tables. In *Proceedings of the 4th International Workshop on Peer-To-Peer Systems, New York, USA*, Februar 2005.

[BO92] Batory, D.; O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Band 1, Nr. 4, Oktober 1992.

[BSR03] Batory, D.; Sarvela, J.; Rauschmayer, A.: Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon*, Mai 2003.

[CC97] Carter, R. L.; Crovella, M. E.: Server Selection Using Dynamic Path Characterization in Wide-Area Networks. In *Proceedings of the 16th IEEE INFOCOM - The Conference on Computer Communications, Kobe, Japan*, April 1997.

[CC04] Colyer, A.; Clement, A.: Large-scale AOSD for middleware. In *Proceedings of the 3rd International Conference on Aspect-oriented software development, Lancaster, UK*, März 2004.

[CD98a] Caro, G.; Dorigo, M.: Ant Colonies for Adaptive Routing in Packet-Switched Communications Networks. In *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, Amsterdam, Holland*, September 1998.

[CD98b] Caro, G.; Dorigo, M.: Two Ant Colony Algorithms for Best-Effort Routing in Datagram Networks. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems, Las Vegas, Nevada*, Oktober 1998.

[CDC+04] Canright, G.; Deutsch, A.; Caro, G.; Ducatelle, F.; Ganguly, N.; Heegaard, P.; Jelasity, M.; Montemanni, R.: Evaluation plan, Project BISON Deliverable D04. `http://www.cs.unibo.it/bison/deliverables/D04.pdf`, September 2004.

[CDEM03] Canright, G.; Deutsch, A.; Engø-Monsen, K.: Structures and Functions of Dynamic Networks, Project BISON Deliverable D02. `http://www.cs.unibo.it/bison/deliverables/D02.pdf`, Juni 2003.

[CDG+02] Castro, M.; Druschel, P.; Ganesh, A.; Rowstron, A.; Wallach, D. S.: Security for structured peer-to-peer overlay networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation, Boston, Massachusetts, USA*, Dezember 2002.

[CDG+04] Caro, G.; Ducatelle, F.; Ganguly, N.; Heegaard, P.; Jelasity, M.; Montemanni, R.: Models for Basic Services in AHN,P2P Networks, Project BISON Deliverable D05. `http://www.cs.unibo.it/bison/deliverables/D05.pdf`, Januar 2004.

[CDJD03] Canright, G.; Deutsch, A.; Jelasity, M.; Ducatelle, F.: Structures and Functions of Dynamic Networks, Project BISON Deliverable D01. `http://www.cs.unibo.it/bison/deliverables/D01.pdf`, Juni 2003.

[CGM99] Cugola, G.; Ghezzi, C.; Monga, M.: Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming. In *Workshop su Sistemi Distribuiti: Algoritmi, Architetture e Linguaggi, L'Aquila, Italy*, September 1999.

[CWH98] Campione, M.; Walrath, K.; Huml, A.: *The Java Tutorial Continued: The Rest of the JDK*. Addison-Wesley, Boston, USA, Dezember 1998.

[DGC+04] Deutsch, A.; Ganguly, N.; Canright, G.; Jelasity, M.; Engø-Monsen, K.: Models for Advanced Services in AHN,P2P Networks, Project BISON Deliverable D08. `http://www.cs.unibo.it/bison/deliverables/D08.pdf`, Januar 2004.

[DGMY03] Daswani, N.; Garcia-Molina, H.; Yang, B.: Open Problems in Data-Sharing Peer-to-Peer Systems. In *Proceedings of the 9th International Conference on Database Theory, Seina, Italy*, Januar 2003.

[Dij74] Dijkstra, E. W.: On the role of scientific thought. published as [Dij82], August 1974.

[Dij76] Dijkstra, E. W.: *A Discipline of Programming*. Prentice Hall PTR, Englewood Cliffs, NJ, Januar 1976.

[Dij82] Dijkstra, E. W.: On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, April 1982.

[DKK+01] Dabek, F.; Kaashoek, F.; Karger, D.; Morris, R.; Stoica, I.: Wide-area Cooperative Storage. In *Proceedings of the 18th ACM Symposium on Operating System Principles, Banff, Canada*, Oktober 2001.

[DMS03] Dingledine, R.; Mathewson, N.; Syverson, P.: Reputation in P2P Anonymity Systems. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems, Berkeley, California, USA*, Juni 2003.

[GC00] Genova, Z.; Christensen, K. J.: Challenges in URL Switching for Implementing Globally Distributed Web Sites. In *Proceedings of the 2000 International Workshop on Parallel Processing, Toronto, Canada*, August 2000.

[GLS+04] Godfrey, B.; Lakshminarayanan, K.; Surana, S.; Karp, R.; Stoica, I.: Load Balancing in Dynamic Structured P2P Systems. In *Proceedings of the 23rd Conference of the IEEE Communications Society, Hong Kong*, März 2004.

[HLM+03] Hausheer, D.; Liebau, N.; Mauthe, A.; Steinmetz, R.; Stiller, B.: Token-based Accounting and Distributed Pricing to Introduce Market Mechanisms in a Peer-to-Peer File Sharing Scenario. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing, Linköping, Schweden*, September 2003.

[HW04] Hsiao, R.; Wang, S.-D.: Jelly: A Dynamic Hierarchical P2P Overlay Network with Load Balance and Locality. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W4: MDC, Hachioji, Tokyo, Japan*, März 2004.

[JMB03] Jelasity, M.; Montresor, A.; Babaoglu, O.: A Modular Paradigm for Building Self-Organizing Peer-to-Peer Applications. In *Proceedings of the 1st International Workshop on Engineering Self-Organizing Applications, Melbourne, Australia*, Juli 2003.

[Knö03] Knöfel, S.: Lastverteilung in Peer-To-Peer Architekturen. Diplomarbeit, Technische Universität Dresden, Fakultät Informatik, Institut Systemarchitektur, Lehrstuhl Betriebssysteme, November 2003.

[KSGM03] Kamvar, S. D.; Schlosser, M. T.; Garcia-Molina, H.: The EigenTrust Algorithm for Reputation Management in P2P Networks. In *Proceedings of the Twelfth International World Wide Web Conference, Budapest, Ungarn*, Mai 2003.

[KSTT04] Krishnan, R.; Smith, M.; Tang, Z.; Telang, R.: The Impact of Free-Riding on Peer-to-Peer Networks. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences, Hawaii, USA*, Januar 2004.

[LB04] Liu, J.; Batory, D.: Automatic Remodularization and Optimized Synthesis of Product-Families. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering, Vancouver, Kanada*, Oktober 2004.

[LKO+00] Lee, M. L.; Kitsuregawa, M.; Ooi, B. C.; Tan, K.-L.; Mondal, A.: Towards self-tuning data placement in parallel database systems. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, Dallas, Texas, USA*, Mai 2000.

[MKL+05] Milojicic, D. S.; Kalogeraki, V.; Lukose, R.; Nagaraja, K.; Pruyne, J.; Richard, B.; Rollins, S.; Xu, Z.: Peer-to-Peer Computing. Technischer Bericht, HP Laboratories, Palo Alto, California, USA, März 2005.

[MO04] Mezini, M.; Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the 12th ACM SIGSOFT, Newport Beach, California, USA*, November 2004.

[Mon04] Monga, M.: On Aspect-Oriented Approaches. In *Proceedings of the European Interactive Workshop on Aspects in Software, Berlin, Germany*, September 2004.

[NOT02] Ng, W. S.; Ooi, B. C.; Tan, K.-L.: BestPeer: A Self-Configurable Peer-to-Peer System. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering (ICDE'02), Boston, USA*, März 2002.

[Ora01] Oram, A.: *Peer-to-Peer*. O'Reilly, United Kingdom, März 2001.

[Par76] Parnas, D. L.: On the Design and Development of Program Families. *IEEE Trans. Software Eng.*, Band 2, Nr. 1, März 1976.

[PB02] Putrycz, E.; Bernard, G.: Using Aspect Oriented Programming to Build a Portable Load Balancing Service. In *Proceedings of the 22nd International Conference on Distributed Computing Systems, Wien, Österreich*, Juli 2002.

[PK99] Poledna, S.; Kroiss, G.: TTP: „Drive by Wire“ in greifbarer Nähe. *Elektronik*, Band 14, Juli 1999.

[RB02] Roussopoulos, M.; Baker, M.: Practical Load Balancing for Content Requests in Peer-to-Peer Networks. *The Computing Research Repository (CoRR)*, Band cs.NI/0209023, September 2002.

[RFH+01] Rantnasamy, S.; Francis, P.; Handley, M.; Karp, R.; Shenker, S.: A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM, San Diego, California, USA*, August 2001.

[RL03] Ramaswamy, L.; Liu, L.: Free Riding: A New Challenge to Peer-to-Peer File Sharing Systems. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences, Hawaii, USA*, Januar 2003.

[RLS+03] Rao, A.; Lakshminarayanan, K.; Surana, S.; Karp, R.; Stoica, I.: Load Balancing in Structured P2P Systems. In *Proceedings of the 2nd International Workshop on Peer-To-Peer Systems, Berkeley, California, USA*, Februar 2003.

[RPW04] Rieche, S.; Petrak, L.; Wehrle, K.: A Thermal-Dissipation-based Approach for Balancing Data Load in Distributed Hash Tables. In *Proceedings of LCN 2004 - 29th IEEE Conference on Local Computer Networks, Tampa, USA*, November 2004.

[SD02] Senior, M.; Deters, R.: Market Structures in Peer Computation Sharing. In *Proceedings of the 2nd International Conference on Peer-to-Peer Computing, Linköping, Schweden*, September 2002.

[SFT02] Schoder, D.; Fischbach, K.; Teichmann, R.: *Peer-to-Peer*. Springer Verlag, Berlin, August 2002.

[SHS05] Saake, G.; Heuer, A.; Sattler, K.-U.: *Datenbanken: Implementierungstechniken*. MITP-Verlag, Bonn, Februar 2005.

[SM02] Sit, E.; Morris, R.: Security Considerations for Peer-to-Peer Distributed Hash Tables. In *Proceedings of the First International Workshop on Peer-to-Peer Systems, Cambridge, Massachusetts, USA*, März 2002.

[SMK+01] Stoica, I.; Morris, R.; Karger, D.; Kaasoek, F.; Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM, San Diego, USA*, August 2001.

[Spi02] Spinczyk, O.: *Aspektorientierung und Programmfamilien im Betriebssystembau*. Dissertation, Fakultät für Informatik, Universität Magdeburg, Dezember 2002.

[SW04] Steinmetz, R.; Wehrle, K.: Peer-to-Peer-Networking &-Computing. *Informatik Spektrum*, Band 27, Nr. 1, Februar 2004.

[Tan03] Tanenbaum, A. S.: *Computer Networks*. Prentice Hall PTR, Englewood Cliffs, NJ, 2003.

[VBW98] Vingralek, R.; Breitbart, Y.; Weikum, G.: Snowball: Scalable storage on networks of workstations with balanced load. *Distributed and Parallel Databases*, Band 6, Nr. 2, April 1998.

[VS05] Voelker, G.; Shenker, S.: *Peer-to-Peer Systems 3*. Springer Verlag, Heidelberg, Januar 2005.

[ZYZ+98] Zhu, H.; Yang, T.; Zheng, Q.; Watson, D.; Ibarra, O. H.; Smith, T.: Adaptive Load Sharing for Clustered Digital Library Servers. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, Illinois, USA*, Juli 1998.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 30. Juli 2005

Karl-Heinz Deutinger