Diplomarbeit

Implementation eines Resourcebrokers im Kontext eines Grid-Datenmanagement-Systems

Humboldt-Universität zu Berlin (HUB)
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
Computer Science Research

eingereicht von: Tom Lichtenstein
Matrikelnummer 173577

Betreuer: Prof. Dr. Alexander Reinefeld (ZIB u. HUB)
Prof. Dr. Miroslaw Malek (HUB)

Berlin, 21. November 2005

Selbständigkeitserklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Berlin, 21. November 2005

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, 21. November 2005

Hinweis zu URLs

Alle in dieser Arbeit angegeben URLs waren zum Zeitpunkt der Fertigstellung dieser Arbeit gültig und enthielten die erwähnten Inhalte. Auf Grund der dynamischen Struktur des Internets können sich die Inhalte mit der Zeit jedoch verändern bzw. vollständig verschwinden.

Danksagung

Danken möchte ich Prof. Dr. Alexander Reinefeld, Leiter des Bereichs Computer Science am Konrad-Zuse-Zentrum für Informationstechnik Berlin und Professor für Parallele und Verteilte Systeme am Institut für Informatik der Humboldt-Universität zu Berlin, und Prof. Dr. Miroslaw Malek, Professor für Rechnerorganisation und Kommunikation am Institut für Informatik der Humboldt-Universität zu Berlin. Ich danke Florian Schintke und Thomas Röblitz vom ZIB für die Betreuung meiner Diplomarbeit. Weiterer Dank gebührt Alain Roy, Ph.D., stellvertretend für das gesamte Condor-Team, der mir all meine Fragen zu Condor sehr ausführlich beantwortet hat, sowie Monika Moser vom ZIB für die Unterstützung bei den notwendigen Erweiterungen am ZIBDMS.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Diese Arbeit ist im Gebiet des *Grid Computing* einzuordnen. Vereinfacht handelt es sich hierbei um eine Technik, die es ermöglicht, verteilte Ressourcen, wie zum Beispiel Rechner, so zu nutzen, als würde es sich bei dem System um einen einzigen Supercomputer handeln [18]. Basierend auf der starken Verbreitung des Internets verbunden mit der Leistungsfähigkeit heutiger Rechner und deren Netzwerkverbindungen ändert sich die Art und Weise, wie Computern benutzt werden. Vor allem durch die immer höhere Bandbreite, die zwischen einzelnen Rechnern verfügbar ist, wird es möglich, Anwendungen inklusive den benötigten Daten auf andere, leistungsfähigere Rechner zu übertragen und dort ausführen zu lassen.

Der Begriff des „Grids“ ist in der Mitte der neunziger Jahre als Bezeichnung für verteilte Systeme zur Lösung großer, computergestützter und meist wissenschaftlicher Probleme entstanden. Ein verteiltes System besteht aus einer Menge unabhängiger Rechner, über denen in einer Middlewareschicht zwischen den Rechnern und den Anwendungsprogrammen eine spezielle Software existiert [39]. Verteilte Systeme sind meist komponentenbasiert aufgebaut und sollten u.a. ständig verfügbar sein und eine hohe Skalierbarkeit bieten.

Eine Aufgabe von Grids ist das Verwalten von heterogenen und oftmals geographisch verteilten Ressourcen. Hierzu gehören Rechner aber auch Programme, Daten oder Messinstrumente, welche bei der Lösung von sehr komplexen Problemen in den Bereichen Forschung, Entwicklung und Handel helfen [1, 45]. Die Verwaltung umfasst die Suche nach Ressourcen, sowie das Sammeln und Zurverfügungstellen von Informationen über diese. Grid Computing unterstützt diese Verwaltungsaufgaben durch die Bildung von dynamischen *Virtuellen Organisationen* [17]. Unter Virtuellen Organisationen versteht man die Zusammenfassung von (Forscher-)Gruppen mit ähnlichen Themenschwerpunkten.

Des Weiteren ermöglichen Grids die gemeinsame Nutzung dieser Ressourcen durch mehrere Anwender. Sie unterstützen den Anwender bei der differenzierten Auswahl für die jeweiligen Anwendungen am besten geeigneten Ressourcen. Weitere Funktionen, die ein Grid bieten sollte, sind die Überwachung eines aktuell ausgeführten *Jobs*, sowie die Übertragung von Ergebnissen von der ausführenden Ressource zurück zum Nutzer [5]. Ein Job bezeichnet hierbei ein Anwendungsprogramm, dessen Ausführung ein Nutzer in Auftrag gegeben hat, und alle dafür benötigten Eingabedaten und Programmparameter.

Grids können in drei Klassen eingeteilt werden [33]: *Information Grids*, *Service Grids* und *Resource Grids*.

Information Grids stellen hauptsächlich Daten zur Verfügung. Der Zugriff auf diese kann i.d.R. kostenlos und anonym erfolgen. Ein Beispiel dafür ist das *World Wide Web*.

Resource Grids ermöglichen den koordinierten Zugriff auf diverse Ressourcen. Diese Ressourcen sind in der Regel nicht kostenlos und nur für authorisierte Benutzer zugänglich. Ziel eines solchen Grids ist es, den Zugriff auf Ressourcen einfach, effizient und transparent zu gestalten. Resource Grids können nochmals in *Computational Grids* und *Data Grids* unterteilt werden. Erstere kümmern sich vor allem um den Zugriff und die Zuteilung von Rechenzeit auf verteilten Supercomputern um zeitintensive Jobs auszuführen. Data Grids bieten Mechanismen für ein sicheres und redundantes Speichern von Ressourcen an, einschließlich Lösungen für Probleme in den Gebieten Replikation, Leistungssteigerung und Caching.

Service Grids dienen dem Nutzer und anderen Anwendungen mit Diensten und Anwendungen unabhängig von ihrem Speicherort, ihrer Implementation und ihrer Hardwareplattform und bauen auf den Ressourcen eines Resource Grids auf.

Diese Arbeit beschäftigt sich hauptsächlich mit Resource Grids und dabei mit Problemen, die sowohl einen effizienten Zugriff auf große Datenmengen als auch eine schnelle Berechnung von Ergebnissen ermöglichen müssen.

Ein Beispiel für ein solches Grid soll im Rahmen des Forschungsprojekts *C3-Grid*[1] [8] entstehen. Dieses im September 2005 beginnende Vorhaben hat als Ziel die „Entwicklung einer hochproduktiven grid-basierten Umgebung für die deutsche Erdsystemforschungsgemeinschaft“. Deren Arbeitsgebiet, die Erdsystemmodellierung, versucht, das System Erde zu verstehen und möglichst zuverlässige Klimavorhersagen zu treffen. Dies ist mit sehr langen Simulationsläufen auf Supercomputern zur Auswertung von hochvolumigen und geographisch verteilten Datensätzen verbunden. Aufbauend auf existierenden Technologien soll im C3-Grid eine „neue Generation eines kollaborativen Gesamtsystems“ für das C3-Grid-Konsortium entwickelt werden. Die Ausgangslage des C3-Grid-Projekts ist ein gutes Beispiel für die Notwendigkeit eines Grid:

1. Es gibt eine Menge von Forschergruppen in geographisch verteilten Klimadatenzentren und anderen Organisationen. Jede Gruppe sammelt und speichert ihre eigenen Daten. Obwohl die Forscher vornehmlich auf ihre eigenen Daten zugreifen, besteht zumindest zeitweise ein Interesse an der Nutzung der Daten anderer Forschergruppen. Eine zentrale Datenhaltung empfiehlt sich nicht, da diese heute und auch in Zukunft technisch nicht praktikabel ist und die Daten in der Vergangenheit auch dezentral erfasst und gespeichert wurden.

[1] Collaborative Climate Community Data and Processing Grid, http://www.c3-grid.de/

2. Es müssen Berechnungen durchgeführt werden, die die Benutzung von Supercomputern notwendig machen, die jedoch nicht allen Forschergruppen bzw. nicht im notwendigen Umfang zur Verfügung stehen.
3. Es gibt große Datenmengen, die für Simulationsläufe zur Verfügung stehen müssen, meist aber getrennt von den Supercomputern vorliegen. So wird für das Jahr 2006 für einige Datenzentren ein Datenvolumen im Petabyte-Bereich und ein Transfervolumen von mehreren 100 Gigabytes pro Monat erwartet.

Zusammengefasst gibt es eine Vielzahl verteilter Datenspeicher, eine Menge heterogener (Super-)Computer, die für Berechnungen zur Verfügung stehen sowie eine Menge von Berechnungsaufträgen, für die bestimmte Daten und bestimmte Rechner benötigt werden. Die Wahl der Ressourcen sollte dabei möglichst automatisch und ohne Interaktion des Nutzers geschehen.

Für diese Aufgaben bietet sich der Einsatz eines Grids an, da es die dafür notwendige Infrastruktur liefert. Diese Arbeit beschäftigt sich im Folgenden hauptsächlich mit der Komponente, die für das Suchen von Ressourcen mit optimaler Eignung für einen bestimmten Job verantwortlich ist, dem so genannten *Resourcebroker*. Meist werden zwei verschiedene Ressourcen für einen Job benötigt, zum einen ein Computer, auf dem die Berechnungen durchgeführt werden können, zum anderen ein Datenspeicher, der die benötigten Daten vorhält und in der Lage ist, diese zum ausführenden Rechner zu schicken. In der Regel handelt es sich dabei nicht um ein und dasselbe Gerät, da Datenspeicher normalerweise nicht die Leistungsfähigkeit besitzen, um die Berechnungen in einer akzeptablen Zeit auszuführen und zum anderen die Rechner nicht genug Kapazität für ein permanentes Speichern der Daten zur Verfügung stellen. Die Ausführbarkeit und die Laufzeit eines Jobs hängen daher ab

- von der Verfügbarkeit eines Datenspeichers mit den benötigten Daten,
- von der Dauer des Datentransfers zum ausführenden Rechner (dies ist wiederum abhängig von der verfügbaren Bandbreite),
- von der Verfügbarkeit eines ausführenden Rechners und der Zeit zur Durchführung der Berechnung und
- von der Zeit zum Zurückschreiben der Ergebnisse bzw. zum Kopieren an den Auftraggeber.

Hauptziel eines Brokers sollte es daher sein, Datenspeicher und ausführenden Rechner so zu wählen, dass die Zeit zwischen dem Beginn des Datentransfers und dem Ende der Berechnungen möglichst minimiert wird. Dazu wurden in der vorangegangenen Studienarbeit „Job-Platzierung im Grid in Abhängigkeit der Replikationsorte der zu verwendenden Daten“ verschiedene Broker [3, 4, 9, 21, 40, 45] untersucht.

Der *Gridbus Broker* hat sich im Vergleich als am besten geeignet erwiesen. Sein Algorithmus wählt für eine Datenquelle die Ressource aus, die den Job am schnellsten ausführen kann. Darin einbezogen ist die Zeit für den Datentransfer sowie die erwartete Ausführungszeit auf der Ressource. Zur Ausführung kommt das Datenquelle-Ressourcen-Paar mit der geringsten Gesamtzeit. Der Nachteil dieses Brokers ist, dass je Job nur eine Datenquelle benutzt wird. Diese muss dann alle benötigten Dateien besitzen. Häufig werden aber Daten von verschiedenen Speicherorten benötigt. Im oben erwähnten Beispiel des C3-Grids soll es den Forschern gerade durch die Kombination eigener und fremder Daten möglich sein, neue Erkenntnisse zu erzielen. Es kann also davon ausgegangen werden, dass häufig nicht alle für einen Job benötigten Daten an einem einzigen Speicherort zu finden sind.

Ein anderes System namens Condor bietet in der aktuellen Version keine Möglichkeit, die Verteilung der benötigten Daten mit in die Ressourcenauswahl mit aufzunehmen. Der Broker von Condor kann zwei Typen unterscheiden, Jobs und Ressourcen. Die Beschreibungssprache für die einzelnen Komponenten und der Broker an sich sind jedoch sehr flexibel gestaltet und nicht auf bestimmte Typen festgelegt. In der zuvor erwähnten Studienarbeit wurde daher die Idee eines dritten Typs für Datenquellen aufgeworfen, um die Verteilung von Daten mit in die Ressourcenauswahl zu integrieren. Die Idee soll in dieser Arbeit näher untersucht und umgesetzt werden.

1.2 Ziel der Arbeit

Hauptanliegen des zu entwickelnden Brokers ist es, Jobs im Grid zu planen und dabei die Verteilung der benötigten Daten mit in die Auswahl einzubeziehen. Dies erfordert eine Zusammenarbeit zwischen dem Datenmanagement, der Jobverwaltung und einem Datentransferdienst, wie sie auch im C3-Grid geplant ist.

Zunächst wird ein Modell für einen Resourcebroker entwickelt, der in der Lage ist, Ressourcen zu suchen, die für einen Job geeignet sind. Hierbei soll zusätzlich beachtet werden, wo die für den Job benötigten Dateien gespeichert sind und mit welchem Aufwand diese zum ausführenden Rechner transportiert werden können. Die Verwaltung der Dateien und ihrer Speicherorte soll nicht durch den Broker selbst erfolgen. Vielmehr wird ein Datenmanagementsystem in das System integriert werden, welches in der Lage ist, Dateitransfers durchzuführen und die dafür benötigten Zeiten zu schätzen.

Der Auftraggeber eines Jobs soll die Möglichkeit haben, zwischen verschiedenen Optimierungsarten zu wählen bzw. bestimmte Grenzen festzulegen, innerhalb welcher der Job ausgeführt werden soll. Beispielsweise kann der Broker ver-

suchen, die Gesamtausführungszeit, also die Zeit vom Beginn des ersten Datentransfers bis zum Ende der eigentlichen Jobausführung, zu minimieren. Es wird auch die Möglichkeit geben, dass der Auftraggeber eines Jobs einen Zeitpunkt festlegt, zu dem der Job abgeschlossen sein muss (*Deadline*). Entsprechend kann auch versucht werden, die Gesamtkosten zu minimieren bzw. ein vorbestimmtest Maximum nicht zu überschreiten.

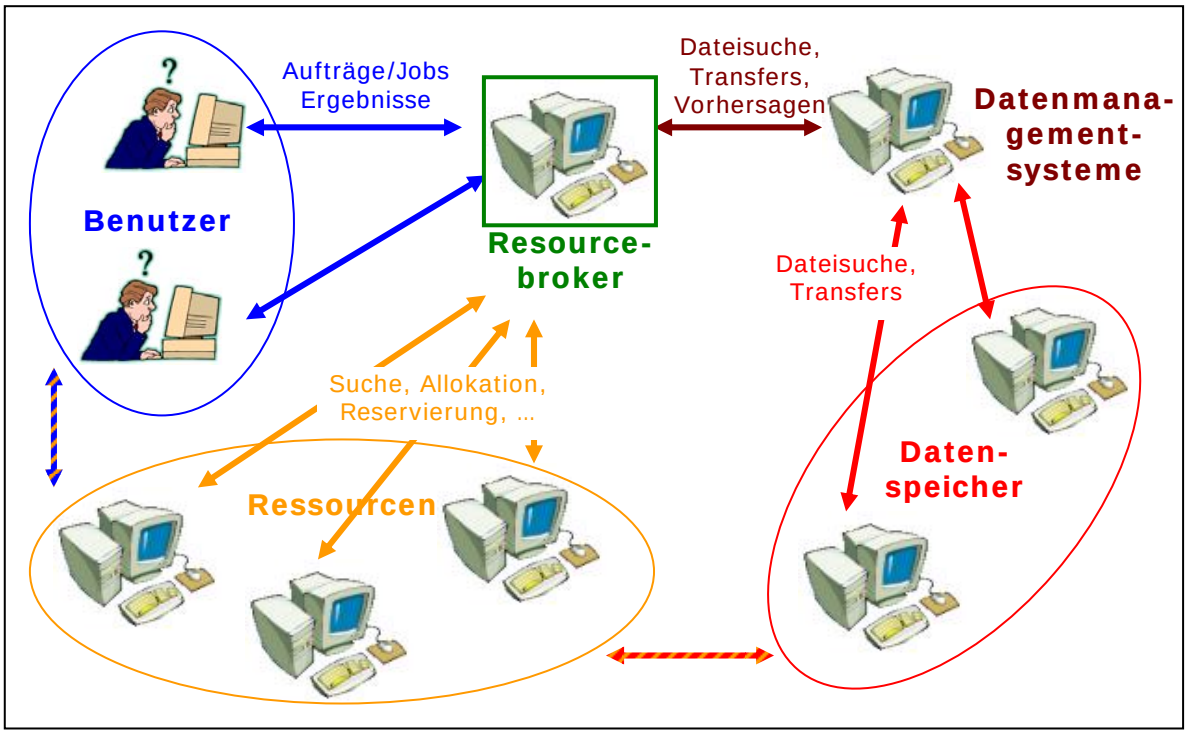


Abb. 1 Kommunikation des Resourcebroker mit beteiligten Komponenten
Zu erkennen sind drei große Gruppen von Komponenten – Benutzer, Ressourcen und Datenspeicher, wobei die letzen über DMSe mit dem Broker kommunizieren. Die schraffierten Pfeile bezeichnen Kommunikation zwischen den Komponenten, die für und während der Jobausführung notwendig sind.

Auf der Basis dieses Modells und dem ClassAd-Prinzip von Condor wird anschließend ein Resourcebroker erstellt und implementiert. Zentrale Komponente wird der Resourcebroker selbst sein, der mit verschiedenen externen Komponenten – Benutzern bzw. deren Rechnern, Ressourcen und Datenspeichern – kommunizieren muss, wie es in Abb. 1 dargestellt ist. Der Broker ist dabei nicht in der Lage, Datenspeicher direkt ansprechen, sondern benutzt zur Informationsgewinnung und zur Übertragung von Dateien die Funktionen von Datenmanagementsystemen. Spezielle Clients dienen als Schnittstelle zwischen den einzelnen Komponenten und dem Resourcebroker. Der Broker sammelt zunächst alle Informationen, die er von den Ressourcen und Datenmanagementsystemen periodisch erhält. Empfängt der Broker eine Jobanfrage, so sucht er in den verfügbaren Quellen nach geeigneten Ressourcen und Datenspeichern. Wird eine Kombination dieser ausfindig gemacht, die die Anforderungen des Auftrags erfüllt, so

wird der Auftraggeber darüber informiert. Anschließend kann sich dieser bzw. dessen Anwendungsprogramm an die jeweiligen Ressourcen wenden, um den Job auszuführen.

Die Beschreibung von Jobs, Ressourcen und Datenspeichern soll mit ClassAds erfolgen, einer bei Condor benutzten Beschreibungssprache. Die Suche passender Ressourcen und Datenspeicher zu einem Job soll nach einem ähnlichen Verfahren wie im Condor-System erfolgen. Dort wird zu einem Job immer eine passende Ressource gesucht. Dieses Verfahren so soll erweitert werden, dass gleichzeitig auch noch Datenspeicher gesucht werden, die die benötigten Daten besitzen.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in die folgenden Abschnitte: In Kapitel 2 wird der Begriff des Resourcebrokers definiert und ein Beispiel für ein solches System vorgestellt. Kapitel 3 präsentiert ein Modell für einen Resourcebroker, der die Verteilung von Daten mit berücksichtigt. In den beiden darauf folgenden Kapiteln 4 und 5 wird die ClassAd-Beschreibungssprache des Condor Systems und das Datenmanagementsystem ZIBDMS vorgestellt, auf welchen der zu entwickelnde Broker aufbaut. Kapitel 6 beschreibt Delphoi, einen Dienst zur Abfrage und Vorhersage von Informationen über Rechner und Netzwerke, der vom ZIBDMS genutzt wird. Kapitel 7 zeigt verwandte Projekte auf Basis von ClassAds auf und grenzt diese von der vorliegenden Arbeit ab. Anschließend wird die Architektur für den Broker (Kapitel 8) und dessen Implementation (Kapitel 9) beschrieben. In Kapitel 10 werden Ergebnisse der Leistungsmessungen an dem neuen Resourcebroker vorgestellt. Eine Zusammenfassung der Arbeit erfolgt in Kapitel 11.

2 Resourcebroker

In diesem Kapitel wird zunächst der Begriff des Resourcebrokers definiert. Im zweiten Abschnitt wird einer der in vorangegangenen Studienarbeit betrachteten Broker zum besseren Verständnis des Themas vorgestellt.

2.1 Definition eines Resourcebrokers

Um einen Job, also eine Berechnung, in einem Grid auszuführen, werden i.d.R. drei Komponenten benötigt:

- eine ***Jobbeschreibung***, also Angaben darüber, welches Programm ausgeführt werden soll, welche Bedingungen dabei erfüllt werden müssen und welche Daten benötigt werden,
- mindestens eine ***Ressource*** (meist ein Rechner), auf dem das Programm ausgeführt werden soll und
- eine *Menge von* ***Daten***, die zur Berechnung nötig sind und die sich meist nicht auf der ausführenden Ressource befinden. De facto können Daten auch als eine Art von Ressourcen betrachtet werden. Aufgrund der Art der in dieser Arbeit betrachteten Jobs empfiehlt es sich aber, Daten und andere Ressourcen getrennt voneinander zu betrachten.

Beim Grid Computing stellt sich das Problem, dass es zum einen eine Vielzahl an Ressourcen gibt, die aber nicht unbedingt alle für die Aufgabe geeignet sind und zudem häufig mit anderen Nutzern geteilt werden müssen. Zum anderen sind die meisten Daten mehrfach im Netz vorhanden, um so Engpässe beim Zugriff durch mehrere Nutzer zu vermeiden. Durch diese Replikate sollen die Transferzeiten vom Datenspeicher zur ausführenden Ressource verkürzt werden. Problematisch ist aber nicht die Vielzahl der Ressourcen und Daten sondern eher die Frage, welche zu nutzen sind, um – aus Nutzersicht – möglichst schnell bzw. kostengünstig zu Resultaten zu gelangen oder – aus Sicht des Netzes bzw. der Ressourcenbesitzer – eine möglichst gleichmäßige Auslastung zu erreichen.

Genau an dieser Stelle setzen *Resourcebroker* an. Resourcebroker sind Softwarekomponenten der Grid-Middleware [9], die zwischen Besitzern von Ressourcen („Produzenten“) und den Nutzern, welche Jobs ausführen wollen („Konsumenten“) vermitteln [7]. Die Besitzer können entscheiden, wer und in welchem Ausmaß Zugriff zu ihren Ressourcen erhält und Kosten für die Nutzung der Ressource festlegen. Konsumenten hingegen haben bestimmte Anforderungen an die Ausführung ihres Jobs, etwa die maximale Bearbeitungszeit (*Deadline*) oder die maximale Höhe des Betrags, den sie zu zahlen bereit sind.

Resourcebroker wandeln die durch einen Nutzer gestellten Anforderungen und Beschränkungen in eine Menge von Jobs um. Diese Jobs werden dann passenden

Ressourcen zugeordnet. Anschließend veranlasst der Broker die Ausführung des Jobs auf den gewählten Ressourcen, überwacht diese und schickt die Ergebnisse zurück an den Nutzer. Resourcebroker verstecken so die Komplexität von Grids, indem sie für den Nutzer alle nötigen und wichtigen Entscheidungen zur Auswahl von Ressourcen treffen. Darüber hinaus muss der Broker in der Lage sein, benötigte Daten zu finden und diese von einer geeigneten Datenquelle hin zur ausführenden Ressource zu übertragen. Die gewählte Datenquelle sollte auch die beste bzgl. der Verfügbarkeit und der Qualität des Datentransfers sein [45]. Bei der Auswahl von Ressourcen und Daten wird der Resourcebroker daher von Managementsystemen unterstützt. Kapitel 5 geht ausführlicher auf Datenmanagementsysteme ein. Einige existierende Resourcebroker sind in der Lage, für einen Job gleichzeitig mehrere Ressourcen zu reservieren, was auch als *Co-Reservation* bezeichnet wird [35].

[38] gliedert das Platzieren von Jobs in 10 obligatorische und einen optionalen Schritt auf[2]. Zum besseren Verständnis des Ablaufs einer Jobplatzierung werden diese hier zusammengefasst vorgestellt (siehe auch Abb. 2). In der Architekturbeschreibung des zu entwickelnden Brokers in Kapitel 8 werden die einzelnen Brokerkomponenten diesen Schritten zugeordnet.

1. ***Autorisationsfilterung***: Wähle aus der Menge aller Ressourcen die aus, auf die der Auftraggeber des Jobs Zugriff hat.
2. ***Anfoderungsdefinition***: Der Auftraggeber kann eine Menge von Anforderungen festlegen, die bei der Wahl der Ressource eingehalten werden müssen. Diese können sowohl statische als auch dynamische Eigenschaften der Ressource betreffen.
3. ***Ressourcenauswahl auf Anforderungsbasis***: Wähle aus der in Schritt 1 verbliebenen Menge von Ressourcen nur die aus, die die Anforderungen aus Schritt 2 erfüllen. Eigenschaften (sowohl statische als auch dynamische) können direkt bei der Ressource, einem Grid Information Service oder bei Datenmanagementsystemen erfragt werden.
4. ***Sammlung dynamischer Informationen***: Alle Ressourcen, die Schritt 3 genügt haben, könnten den Job ausführen. Aus diesen soll im Folgenden eine möglichst gute Ressource gewählt werden. Daher werden weitere Informationen über die Ressource, etwa deren aktuelle Auslastung, eingeholt.
5. ***Wahl der besten Ressource***: Auf der Basis von Schritt 4 wird die beste Ressource gewählt. Der jeweils eingesetzte Algorithmus unterscheidet sich hier je nach Broker und Anwendungsgebiet.
6. ***Ressourcenreservierung***: Dieser Schritt ist optional und wird von vielen Resourcebrokern nicht unterstützt. Die Reservierung soll einem Job garan-

[2] [38] teilt die Schritte darüber hinaus noch in drei Phasen auf: Ressourcenentdeckung (Schritt 1-3), Ressourcenauswahl (4-5), Jobausführung (6-11).

tieren, dass die Ressource zum Zeitpunkt der Ausführung in der Weise zur Verfügung steht, wie es in den vorherigen Schritten angenommen wurde. Problematisch ist hierbei, dass das Gridsystem in der Regel nicht die volle Kontrolle über eine Ressource hat. Dadurch kann die Ressource durch andere Prozesse ausgelastet sein.

7. ***Jobübermittlung***: Nach der Wahl der Ressource wird die entsprechende Anwendung an die Ressource übertragen.
8. ***Vorbereitung der Ausführung***: Hierzu gehört die Vorbereitung der Umgebung der Anwendung und die Sicherstellung, dass alle benötigten Daten auf der Ressource vorhanden sind. In [38] zwar nicht explizit erwähnt, startet die Ausführung des Jobs nach Abschluss der Vorbereitungen.
9. ***Ausführungsüberwachung***: Die Überwachung dient zum einen dazu, dem Nutzer Informationen über die Ausführung zur Verfügung zu stellen, etwa Fortschrittsangaben. Zum anderen kann das System im Falle eines Fehlers einen Job neu planen und wieder starten.
10. ***Jobabschluss***: Nach Beendigung des Jobs sollte der Auftraggeber über das Ende der Ausführung informiert werden. I.d.R. geschieht dies per E-Mail.
11. ***Aufräumen***: Zuletzt sollten alle Ergebnisdaten an den Auftraggeber zurückgeschickt und alle Dateien, die im Rahmen des Jobs auf der Ressource angelegt wurden, wieder gelöscht werden.

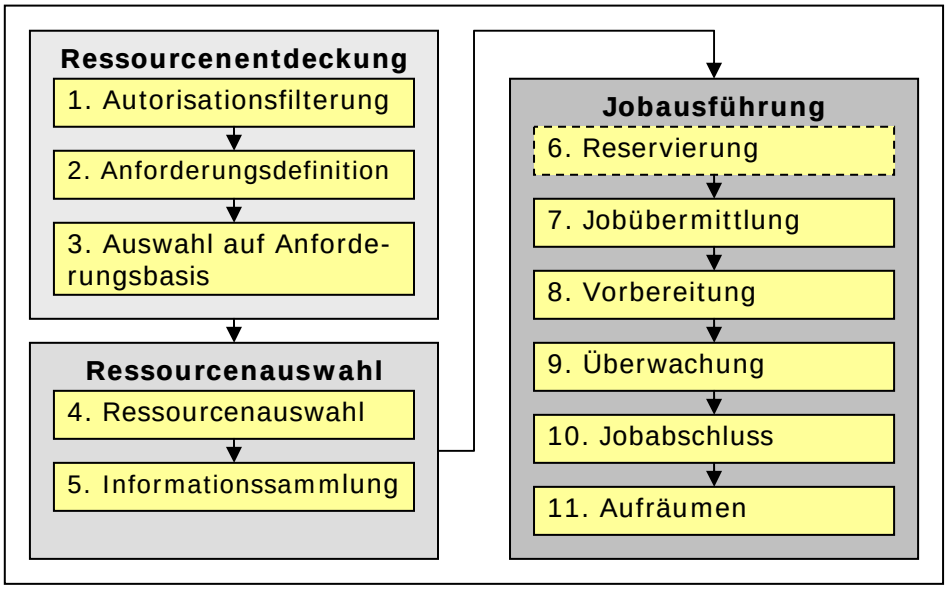


Abb. 2 Die drei Phasen und 11 Schritte der Jobplatzierung im Grid nach [38]

2.2 Der Gridbus Broker – Beispiel für einen Resourcebroker

Der Gridbus Broker [45, 46] der University of Melbourne, Australien, wurde bereits in der vorangegangen Studienarbeit vorgestellt. In dieser wurden Resourcebroker daraufhin verglichen, inwieweit sie die Verteilung und Verfügbarkeit von

Daten mit in die Ressourcenauswahl bei der Planung von Jobs einbeziehen. Im Vergleich hat sich der Gridbus Broker, eine Erweiterung des Nimrod/G-Brokers [2], am besten dafür geeignet gezeigt. Grund dafür ist, dass bei der Suche nach Ressourcen berücksichtigt, wie lange nötige Datentransfers voraussichtlich dauern. Der Nachteil des Resourcebrokers ist, dass alle benötigten Daten auf einem einzigen Datenspeicher vorliegen müssen, wovon in der Regel nicht ausgegangen werden kann. Im C3-Grid wollen die Forscher beispielsweise auf die Daten anderer Organisationen zugreifen. Ein zentraler Datenspeicher ist aber nicht geplant, wodurch sich die für einen Job benötigten Daten nur zufällig an einem einzigen Speicherort befinden werden.

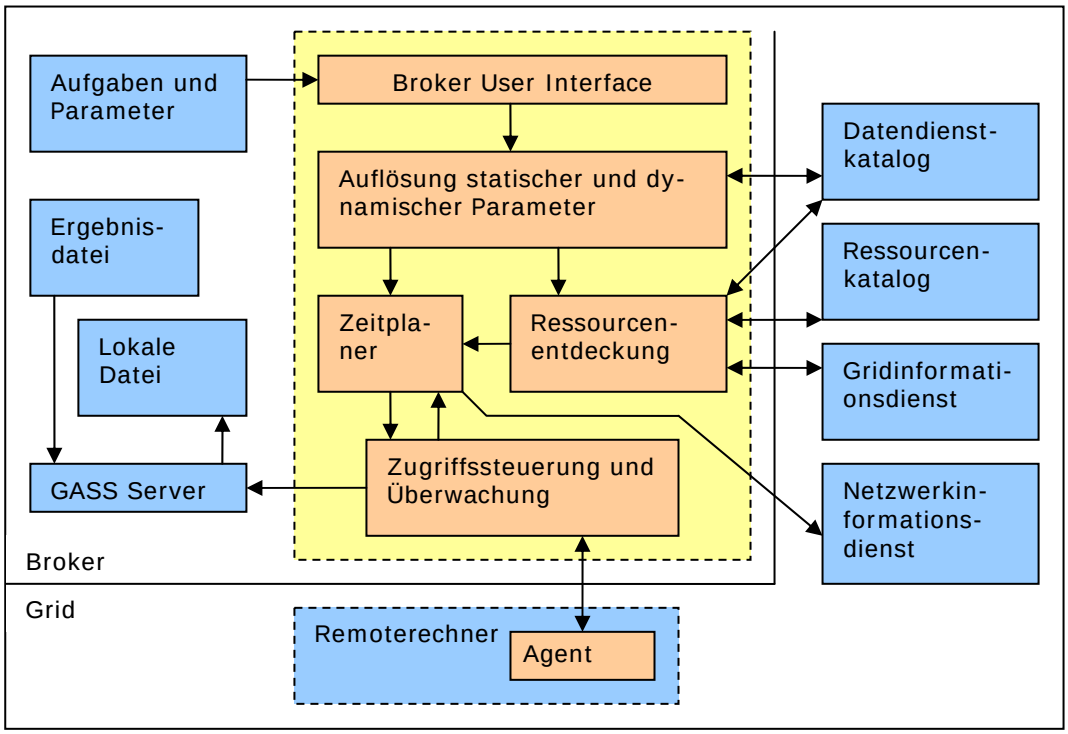


Abb. 3 Architektur des Gridbus Brokers, nach [46]
Der Broker selbst ist hellgrau hinterlegt. Pfeile zu Komponenten außerhalb des Brokers stellt Interaktion mit anderen Gridkomponenten dar.

Der Aufbau des Brokers ist in Abb. 3 dargestellt. „Eingaben“ für den Gridbus Broker sind Aufgaben (*tasks*) und die dazugehörigen Parameter samt ihrer Werte. Als Aufgabe wird eine Menge von Befehlen verstanden, mit denen der Nutzer seine Anforderungen definiert. Die Parameter können sowohl statisch als auch dynamisch sein, wobei dynamische Parameter zur Laufzeit ausgewertet werden. Die Syntax der Eingabedateien zur Beschreibung der Aufgaben entspricht der bei Nimrod/G[3] verwendeten deklarativen Programmiersprache. Diese wurde um einen Parametertyp „Gridfile“ erweitert, einem dynamischen Parameter, der einen logischen Dateinamen, ein Verzeichnis oder eine Sammlung von Dateien enthalten kann, die durch den Broker aufgelöst werden.

[3] http://www.csse.monash.edu.au/~davida/nimrod/nimrodg.htm.

Die Ressourcenentdeckung erfolgt auf Basis der durch den Nutzer in den Tasks festgelegten Anforderungen und mit Hilfe von Informationsdiensten wie dem Globus[4] Grid Index Information Service (GIIS). Zusätzlich zu den durch die Informationsdienste gelieferten verfügbaren Ressourcen kann der Nutzer auch eine Liste verfügbarer Ressourcen angeben. Der Broker kann auch die Informationsdienste jedes Rechnerknoten nutzen, um zusätzliche Eigenschaften der Ressourcen zu erfahren. Logische Dateinamen werden (nach Auswertung eventuell vorhandener dynamischer Parameter) mittels der Replikat- und Datendienstkatalog in physikalische gewandelt.

Anschließend werden aus Aufgabe und Parametern Jobs gebildet, wobei ein Job eine Instanz einer Aufgabe mit einem eindeutigen Parametersatz ist. Die Jobs werden zusammen mit den verfügbaren Ressourcen an den Zeitplaner (*scheduler*) weitergegeben, welcher die Jobs mit den Ressourcen abgleicht und sie an diese übermittelt. Beim Planen von Jobs, die verteilt abgelegte Daten benötigen, interagiert der Zeitplaner mit Netzwerkinformationsdiensten, um die verfügbaren Bandbreiten zwischen Datenquellen und Remoterechnern mit zu berücksichtigen. Der in der aktuellen Version verwendete Network Weather Service misst z.B. TCP-Verbindungszeiten, Latenzzeiten zwischen Verbindungsendpunkten und die Netzwerkbandbreite. Durch die Analyse dieser Daten ist der Dienst in der Lage, Vorhersagen über zukünftige Auslastungen zu erstellen.

Der Scheduler betrachtet das Grid aus Sicht der Daten und minimiert den nötigen Datentransfer während der Jobausführung, indem die ausführenden Ressourcen so gewählt werden, dass sie möglichst dicht bei den Datenquellen liegen. Optimal in diesem Sinne wäre es, die Berechnungen direkt auf der Datenquelle ausführen zu lassen, da es hier quasi eine unendlich große Bandbreite gibt. Nur ist die Rechenkapazität von Datenspeichern oft sehr begrenzt. Der Scheduler muss daher zwischen der verfügbaren Bandbreite und der Kapazität und Leistung der Ressource abwägen.

Dateien können als logische Dateinamen innerhalb einer virtuellen Verzeichnisstruktur organisiert sein, die durch einen Replikat- bzw. Datendienstkatalog verwaltet wird. Bevor ein Job in den Zeitplaner aufgenommen wird, werden mit Hilfe des Datenkatalogs alle Datenquellen gesucht, die eine physische Datei mit dem angegebenen LFN enthalten. Anschließend wird für jede dieser möglichen Datenquellen die Ressource gesucht, die den Job in der kürzesten Zeit ausführen kann – unter Beachtung der aktuelle Auslastung der Ressource, der Zeit zum Datentransfer bei der aktuell verfügbaren Bandbreite und anderer Zustandsinforma-

[4] Ein Grid-Middleware-System, siehe auch: The Globus Alliance, http://www.globus.org

tionen. Der Algorithmus findet sich in Abb. 4. Der Gridbus Broker sucht das Datenspeicher-Ressourcen-Paar, das den Job zum frühest möglichen Zeitpunkt beenden kann. Der Resourcebroker, der in dieser Arbeit entwickelt werden soll, kann darüber hinaus auch die Gesamtausführungszeit minimieren und für jeden Job Daten von mehr als einem Datenspeicher beziehen.

Initialisierung

1. Ermittle Ressourceneigenschaften über Grid Information Services (GIS).
2. Erstelle Aufgabenbeschreibung, bei logischen Dateinamen:
   a. Fordere Liste der logischen Dateinamen beim Dataekatalog an
   b. Ermittle für jeden logischen Dateiname alle physischen Replikate durch Anfragen an den Datenkatalog.

Scheduling-Schleife

Solange es nicht-geplante Aufgaben gibt:

3. Ermittle für jede Ressource die geschätzte Ausführungszeit (über die Job consumpation rate).
4. Gibt es Jobs, die bei einer Ressource aus Ausführung warten und gibt es andere freie Ressourcen, so füge den Job in die Liste der wartenden Jobs hinzu.
5. Solange es nicht zugeordnete Jobs gibt und es Ressourcen freie Kapazitäten besitzen, wiederhole:
   a. Wähle den nächsten Job aus der Warteschlange.
   b. Identifiziere alle Datenspeicher der logischen Dateien des Jobs.
   c. Erstelle eine Datenspeicher-Ressourcen-Liste für den aktuellen Job:
      – Suche für jeden Datenspeicher die Ressource, die den Job in der kürzesten Zeit ausführen kann (inkl. Datentransferzeiten) und speichere das Paar.
   d. Wähle die Kombination aus der Liste mit dem frühesten Jobende.
   e. Gibt es so ein Paar, ordne den Job der Ressource zu und entferne ihn aus der Liste.
6. Ende der Schleife.

Abb. 4 Scheduling-Algorithmus nach [45]

Das Übermitteln der Jobs an die Ressourcen übernimmt die Zugriffssteuerung (*actuator*) unter Nutzung der darunter liegenden Middleware wie z.B. Globus oder Alchemi[5]. Dazu wird der Job in einem Agenten (*agent*) gekapselt, welcher an den Remoterechner übermittelt und dort ausgeführt wird. Der Agent ist in der Lage, benötigte Daten von anderen Datenquellen per *Globus Access to Secondary Storage* (GASS) zu laden um vom Broker bereitgestellte Dateien mit Konfigurationsdaten oder Eingabeparametern zu erhalten. Nach der Ausführung des Jobs übermittelt der Agent alle Ergebnisse zurück an den Broker.

[5] .NET-basierte Grid Computing Plattform für Windowsrechner

3 Modell des datenabhängigen Resourcebrokers

Dieses Kapitel beschreibt ein formales Modell für einen Resourcebroker, der die Verteilung und Verfügbarkeit von Daten bei der Jobauswahl berücksichtigt. Es werden alle relevanten Komponenten des Brokers und zwei Optimierungsansätze vorgestellt.

3.1 Komponenten

Ressourcen sind Dinge, die für eine bestimmte Zeit benutzt werden können. Die Nutzung von Ressourcen kann kostenpflichtig oder kostenlos sein. Ressourcen gehören einem Besitzer, der entscheiden kann, wer Zugriff zu diesen erhält und ob und wie viel dafür zu bezahlen ist. Die Nutzung kann exklusiv oder gemeinsam mit anderen erfolgen. Ressourcen sind in diesem Modell Rechner, auf denen Anwendungen ausgeführt werden können. Sie besitzen bestimmte Eigenschaften, wie Systemarchitektur oder Betriebssystemversion.

- $\mathbf{R} \subseteq \mathbf{IN}$ sei die Menge der im Modell verfügbaren Ressourcen[6]. Für $\mathbf{r} \in \mathbf{R}$ sei dann:
- $\mathbf{cu}^{\mathbf{r}}$ *Kosten pro Nutzung* (Fixkosten[7]) der Ressource in *Wert*[8].
- $\mathbf{ct}^{\mathbf{r}}$ *Kosten pro Zeiteinheit* für die Nutzung einer Ressource in *Wert/Sekunde*.
- $\mathbf{at}^{\mathbf{r}}$ *Anzahl der Aktionen*, die in einer Zeiteinheit ausgeführt werden können in *Befehle/Sekunde*.
- $\mathbf{A} \subseteq \mathbf{IN}$ sei die Menge aller möglichen Eigenschaften (Attribute) aller Ressourcen. Eine Ressource $\mathbf{r} \in \mathbf{R}$ besitzt eine Eigenschaft $\mathbf{a} \in \mathbf{A}$ wenn gilt: $\mathbf{a} \in \mathbf{A}^{\mathbf{r}} \subseteq \mathbf{A}$.

Eine ***Datei*** ist eine spezielle Form einer Ressource. Dateien sind persistente Speicher für inhaltlich zusammengehörende Daten. Eine Datei kann einen oder mehrere logische Namen besitzen und ist auf mindestens einem physischen Datenspeicher (siehe unten) gespeichert. Meist existieren mehrere Replikate einer Datei auf verschiedenen Speichern. Inwiefern es sich bei zwei physischen Datei-

---

6 **IN** sei die Menge der natürlichen Zahlen.

7 Fixkosten bezeichnen in der Mikroökonomik Kosten, die unabhängig vom Outputniveau anfallen. Eine Unterscheidung in Fix- und variable Kosten ermöglicht ein flexibleres Modell. So kann zwischen einer Flatrate (nur Fixkosten), einem grundgebührbasierten System (Fix- und variable Kosten) und einem leistungsabhängigen System (nur variable Kosten) durch eine Änderung der Variablenwerte einfach gewechselt werden. Siehe auch „(Quasi-)Fixkosten“ in [44], Abschnitte 20.4 und 21.1.

8 *Wert* wurde in diesem Modell als Einheit für Kosten gewählt, um eine Abstraktion von konkreten Geldzahlungen zu erreichen. Beispielsweise kann es zwischen verschiedenen Nutzern Ausgleich in Form von Freikontigenten geben, in welche ein Wert umgerechnet werden kann.

en um zwei Replikate der gleichen Datei handelt, ist eine Festlegung des Anwenders und kann nicht durch ein Softwaresystem erkannt werden [10]. In solchen Systemen werden Dateien daher zusammen mit einem eindeutigen Bezeichner abgelegt, über den das System dann die Replikate erkennen kann.

- $\mathbf{D} \subseteq \mathbf{IN}$ sei die Menge der im Modell verfügbaren Dateien.
- $\mathbf{size^d, d} \in \mathbf{D}$ gibt die *Dateigröße* der Datei **d** in *Byte* an.

Ein ***Datenspeicher*** ist eine spezielle Ressource, die vor allem zum Speichern von Replikaten einer Datei benutzt wird. Wie die Erstellung und Verteilung der Replikate erfolgt, ist für das Modell irrelevant. Ein Datenspeicher bietet die Möglichkeit, anzufragen, ob ein Replikat einer Datei auf diesem Datenspeicher vorhanden ist oder nicht. Dies ist notwendig, da es kein globales Wissen über die Verteilung der Replikate gibt.

- $\mathbf{S} \subseteq \mathbf{IN}$ sei die Menge der Datenspeicher im Modell.
- $\mathbf{D^s} \subseteq \mathbf{D, s} \in \mathbf{S}$ sei die Menge der Replikate, die der Datenspeicher **s** besitzt.

Zeitangaben und -spannen können sowohl absolut als auch relativ sein. Die Darstellung der Zeit im Modell erfolgt mittels einer natürlichen Zahl **z**, die für jede Sekunde, die im Modell vergeht, um 1 erhöht wird. Relative Zeitangaben können ebenfalls durch natürliche Zahlen dargestellt werden, wobei eine Zahl **z'** besagt, dass zwischen zwei Zeitpunkten **z'** Sekunden vergangen sind.

In der folgenden Komponente ist es nötig, Vorhersagen über Eigenschaften aufgrund von Erkenntnissen aus der Vergangenheit zu treffen. Dazu muss angenommen werden, dass sich die Werte einem bestimmten Muster entsprechend über die Zeit verändern, wobei sich dieses Muster periodisch wiederholt[9]. Um dies zu modellieren, benötigt man eine Funktion, die eine Zeitangabe in ein Raster einordnet. Sei $\mathbf{T} = \mathbf{\{0,1,...n\}}$ ein solches Raster und $\mathbf{z}' \in \mathbf{IN}$ eine Zeitangabe, so gilt für eine solche Funktion **f**:

- $\mathbf{f : IN \rightarrow T}$

Um z.B. eine Zeitangabe in ein Wochentagsraster einzuordnen, muss eine solche Funktion zunächst die Zeitangabe in eine Tagesangabe umwandeln und diese durch Modulo-Rechnung einem Wochentag zuordnen.

Eine ***Netzwerkverbindung*** ist die Verbindung zwischen einem Datenspeicher und einer Ressource, wobei beide entweder direkt miteinander verbunden sein

[9] Ein Beispiel wäre der Stromverbrauch privater Haushalte. So wird der Stromverbrauch an jedem Werktag ähnlich sein (abgesehen von Auswirkungen durch Wetteränderungen), im Vergleich zum Wochenende wird es jedoch Unterschiede geben. Beim Vergleich ganzer Wochen miteinander werden sich die Daten bei gleich bleibenden Werten wiederum ähneln. Siehe auch [26]

können oder eine Verbindung unter Benutzung anderer Ressourcen hergestellt werden kann. Eine verfügbare Netzwerkverbindung hat eine bestimmte Bandbreite, die je nach Auslastung schwanken kann. Es wird angenommen, dass diese Schwankungen einem etwa gleich bleibenden, sich periodisch wiederholenden Muster unterliegen.

- $\mathbf{n} = \langle \mathbf{s}, \mathbf{r} \rangle, \mathbf{s} \in \mathbf{S}, \mathbf{r} \in \mathbf{R}$ sei eine Netzwerkverbindung.
- $\mathbf{N} \subseteq \mathbf{S} \times \mathbf{R}$ sei die Menge aller verfügbaren Netzwerkverbindungen im Modell.
- $\mathbf{Z} = \{0,1,\ldots,n\}$ sei ein Raster wie es im vorherigen Absatz beschrieben wurde und $\mathbf{f(z)}, \mathbf{z} \in \mathbf{IN}$ eine Funktion, die eine Zeitangabe in das Raster einordnet.
- $\mathbf{B} = \mathbf{N} \times \mathbf{Z}$ sei eine Matrix, die die verfügbare Bandbreite zwischen zwei Ressourcen in Abhängigkeit von der Zeit angibt. Für ein Element $\mathbf{b}_{s,r,z} \in \mathbf{B}, \mathbf{z} \in \mathbf{Z}$ gilt: $\mathbf{b}_{s,r,z} = \begin{cases} \infty, \mathbf{s} = \mathbf{r} \\ \mathbf{x} \in \mathbf{IN}, \mathbf{sonst} \end{cases}$ wobei $\mathbf{x}$ exogen gegeben ist und die erwartete Bandbreite angibt. Die Bandbreite wird in *Byte/Sekunde* angegeben.
- $\mathbf{b(s, r, z)} = \mathbf{b}_{s,r,t'}, \mathbf{t'} = \mathbf{f(z)}$ gibt die Bandbreite für eine absolute Zeit an.
- $\mathbf{cu}^{\mathbf{n}}$ sind die *Kosten je Nutzung* (Fixkosten) einer Netzwerkverbindung in *Wert*.
- $\mathbf{cb}^{\mathbf{n}}$ sind die *Kosten je Byte* in *Wert/Byte.* Die Kosten sind in diesem Modell nicht von der Zeit abhängig (*Flat Price Model*).

Ein ***Dateitransfer*** überträgt eine Datei von einem Datenspeicher zu einer Ressource. Für den Transfer einer Datei muss eine Netzwerkverbindung zwischen der Quelle und dem Ziel existieren und es muss ein Replikat der Datei auf dem Datenspeicher vorhanden sein. Für den Fall, dass es keine Netzwerkverbindung zwischen der Datenquelle und der Ressource gibt, sind die Gesamtkosten und die Gesamtzeit für den Datentransfer unendlich hoch.

Sei $\mathbf{d} \in \mathbf{D}$ eine zu übertragende Datei, $\mathbf{s} \in \mathbf{S}$ ein Datenspeicher und $\mathbf{r} \in \mathbf{R}$ die Ressource, zu der die Datei übertragen werden soll. Dann gibt die Funktion $\mathbf{possible}^{\mathbf{t}}\mathbf{(d, s, r)}$ an, ob ein Datentransfer der Datei $\mathbf{b}$ von $\mathbf{s}$ nach $\mathbf{r}$ möglich ist:

$$- \quad \mathbf{possible}^{\mathbf{t}}\mathbf{(d, s, r)} = \begin{cases} \mathbf{1}, \mathbf{d} \in \mathbf{D}^{\mathbf{s}} \wedge \langle \mathbf{d}, \mathbf{s} \rangle \in \mathbf{N} \\ \mathbf{0}, \mathbf{sonst} \end{cases} \tag{2}$$

Die *Gesamtkosten* eines Datentransfers in *Wert* ergeben sich aus den Fixkosten der Netzwerkverbindung und den Kosten, die von der Datenmenge abhängig sind:

$$- \quad \mathbf{ct}^{\mathbf{t}}\mathbf{(d, s, r)} = \begin{cases} \infty, \mathbf{possible}^{\mathbf{t}}\mathbf{(d, s, r)} = \mathbf{0} \\ \mathbf{cu}^{\langle \mathbf{s}, \mathbf{r} \rangle} + \mathbf{cb}^{\langle \mathbf{s}, \mathbf{r} \rangle} \cdot \mathbf{size}^{\mathbf{d}}, \mathbf{sonst} \end{cases} \tag{3}$$

Die *Gesamtzeit* für einen Datentransfer in *Sekunden* ergibt sich aus der Dateigröße und der verfügbaren Bandbreite, z_s gibt die Startzeit an:

– $$tt^t(d, s, r, z) = \begin{cases} \infty, & possible^t(d, s, r) = 0 \\ size^d / b(s, r, z), & sonst \end{cases} \quad (4)$$

Ein ***Job*** bezeichnet die Ausführung eines Programms auf einer Ressource. Um einen Job abzuarbeiten, muss eine Anzahl von Aktionen ausgeführt werden. Im Modell genügt ein Job, da der Resourcebroker immer versucht, nur für einen Job passende Ressourcen zu finden (*Application Scheduling*).

– a^j sei die Gesamtzahl der Aktionen in *Befehlen*.

Die *Gesamtzeit* der Ausführung eines Jobs auf einer Ressource berechnet sich aus der Anzahl der Aktionen des Jobs und der Anzahl der Aktionen, die in einer Zeiteinheit auf der Ressource verarbeitet werden können. I.d.R. ist die Anzahl der Aktionen eines Jobs abhängig von der Menge der Eingabedaten. In diesem Modell wird dies vereinfacht dargestellt, indem angenommen wird, dass dies bei Festlegung der Anzahl der Aktionen des Jobs mit berücksichtigt wurde.

– $$tt^j(r) = a^j / at^r, r \in R \quad (5)$$

Die *Gesamtkosten* für die Ausführung eines Jobs ergeben sich aus der Summe der fixen Benutzungskosten der Ressource und den Kosten je Zeiteinheit, multipliziert mit der Gesamtausführungszeit.

– $$ct^j(r) = cu^r + ct^r \cdot tt^j, r \in R \quad (6)$$

Damit ein Benutzer einen Job ausführen kann, muss dieser einen ***Jobauftrag*** an den Resourcebroker übermitteln. Ein solcher Auftrag umfasst einen Job, eine Menge von Anforderungen an die ausführende Ressource sowie eine Menge von Dateien, die zur Ausführung des Jobs notwendig sind. Darüber hinaus besitzt jeder Jobauftrag eine Ranking-Funktion zur Auswahl einer Ressource. Es ist möglich, dass es mehr als eine Ressource gibt, die die Anforderungen erfüllt. In diesem Fall wird die Ranking-Funktion benutzt, welche für eine Ressource angibt, inwiefern sie anderen Ressourcen gegenüber bevorzugt wird. I.d.R. wird auf Grundlage der Attributwerte der Ressource ein Ergebnis berechnet.

– $A^j \subseteq A$ sei die Menge der Anforderungen eines Jobs an eine Ressource.
– $D^j \subseteq D$ sei eine Menge von Dateien, die für die Ausführung des Jobs benötigt werden.
– $rank^j(r), r \in R$ sei eine Rankingfunktion. Für eine Ressource mit den Eigenschaften $a_1, a_2 \in A^R$ z.B. $rank^j(r) = a_1 * a_2$.

– $\langle a^j, A^j, D^j, rank^j(r) \rangle$ sei ein Jobauftrag, wobei a^j die Anzahl der Aktionen zur Ausführung des Jobs ist.

Eine ***Lösung*** des Resourcebrokers für einen Jobauftrag $\langle a^j, A^j, D^j, rank^j(r) \rangle$ umfasst zunächst eine Ressource, die die Anforderungen des Auftrags erfüllt:

– $r^*(A^j) \in \{ r \mid r \in R \wedge A^j \subseteq A^r \}$

Für jede benötigte Datei im Jobauftrag muss ein Datenspeicher S^* gefunden werden, der zum einen ein Replikat der Datei besitzt und zum anderen über eine Netzwerkverbindung mit der ausführenden Ressource verbunden ist:

– $S^*(r^*, D^j) =$
$\{ s \mid s \in S \wedge \forall d \in D^j : \exists s' \in S^* : d \in D^{s'} \wedge \langle s', r \rangle \in N \wedge |S^*| = |D^j| \}$

Eine Lösung ist ein Paar aus einer Ressource und einer Menge von Datenspeichern, die die benötigten Dateien vorhalten:

– $L(A^j, D^j) = \langle r^*(A^j), S^*(r^*, D^j) \rangle$ (7)

Aus L lässt sich die Menge der notwendigen Datentransfers folgern:

– $T^L = \{ \langle r, s \rangle \mid \langle r, s' \rangle \in L \wedge s \in S' \}$ (7b)

3.2 Optimierungsproblem

Im Rahmen dieser Arbeit werden zwei verschiedene Optimierungsziele unterschieden – *Gesamtkosten* und *Gesamtausführungszeit*. Ist ersten Fall wird versucht, die Kosten, die für jede Aktion anfallen, so gering wie möglich zu halten. Im zweiten Fall soll die Zeitspanne zwischen dem Beginn des ersten Datentransfers und dem Ende der eigentlichen Jobausführung minimiert werden.

3.2.1 Minimierung der Gesamtkosten

Die Gesamtkosten ct, gegeben ein Jobauftrag $\langle a_j, A^j, D^j, rank^j(r) \rangle$ und eine mögliche Lösung für den Auftrag $L = \langle r^*, S^* \rangle$, ergeben sich durch die Addition der Kosten der einzelnen Datentransfers (3) und der Kosten der Jobausführung (6):

– $$ct(r, D^j, S^*) = ct^j(r) + \sum_{x=1}^{|D^j|} ct(D_x^j, S_x^*, r)$$ (8)

Im Folgenden wird davon ausgegangen, dass nur Datentransfers betrachtet werden, die auch möglich sind, d.h. bei denen (2) als Ergebnis 1 liefert.

– $$ct(r, D^j, S^*) = cu^r + ct^r \cdot tt^j + \sum_{x=1}^{|D^j|} cu^{\langle S_x^*, r \rangle} + cb^{\langle S_x^*, r \rangle} \cdot size^{D_x^j}$$ (9)

Aus (5) und (9) ergibt sich dann:

$$- \quad ct(r, D^j, S^*) = cu^r + ct^r \cdot a^j / at^r + \sum_{x=1}^{|D^j|} cu^{\langle S_x^*, r\rangle} + cb^{\langle S_x^*, r\rangle} \cdot size^{D_x^j} \quad (10)$$

Die Belegungen für r und S^* müssen nun so gewählt werden, dass die Gesamtkosten minimiert werden. D.h. es kann keine andere Belegung $\langle r', S'\rangle$ geben, deren Gesamtkosten geringer sind als die der Belegung $\langle r, S^*\rangle$:

$$- \quad S' = \left\{ s \middle| s \in D \wedge \forall d \in D^j : \exists s' \in S' : d \in D^{s'} \wedge \langle s', r\rangle \in N \wedge |S'| = |D^j| \right\}$$

$$- \quad L^* = \left\{ \langle r, S^*\rangle \middle| \forall r' \in R, r' \neq r, \forall S' \neq S^* : ct(r, D^j, S^*) \leq ct(r', D^j, S') \right\} \quad (11)$$

Vereinfachend wurde in diesem Modell angenommen, dass sich die einzelnen Datentransfers nicht gegenseitig beeinflussen. Findet ein Datentransfer über eine Netzwerkverbindung statt, so kann man davon ausgehen, dass für einen zweiten Datentransfer zur gleichen Zeit über dieselbe Netzwerkverbindung nicht die gleiche Bandbreite zur Verfügung steht. Dies hätte zur Folge, dass eine Nutzung von Netzwerkverbindungen nur über einen Zeitplan möglich wäre und damit die Komplexität des Modells stark steigen würde.

Die Menge L^* aus (11) kann immer noch mehr als ein Element enthalten. Daher muss für jede mögliche Lösung in L^* noch der Rang bestimmt werden. Zur Ausführung kommt die Lösung mit dem größten Rang. Gibt es mehrere Lösungen mit dem gleichen größten Rang, so wird die zuerst gefundene benutzt:

$$- \quad \langle r^{**}, S^{**}\rangle \in L^* \wedge \forall \langle r', S'\rangle \in L, \langle r', S'\rangle \neq \langle r^{**}, S^{**}\rangle : rank^j(r^{**}) \geq rank^j(r') \quad (12)$$

3.2.2 Minimierung der Gesamtzeit

Ziel der Minimierung der Gesamtzeit ist nicht die Minimierung der Summe der einzelnen Aktionen. Dies kann durch Minimierung der Gesamtkosten erreicht werden, wenn die Fixkosten (also die Kosten je Benutzung einer Komponente) gleich 0 gesetzt werden und die Kosten je Zeiteinheit für alle Komponenten gleich sind. Bei der Optimierung der Gesamtzeit geht es vielmehr darum, die Zeitspanne vom Beginn des ersten Datentransfers bis zum Ende der Jobausführung zu minimieren.

Um Aussagen über Zeiten machen zu können, benötigt jeder Datentransfer und jeder Job zunächst eine Start- und eine Endzeit:

- $z_S^j \in \mathbf{IN}$ sei die Startzeit des Jobs $\mathbf{j}$ und $z_S^t \in \mathbf{IN}$ sei die Startzeit eines Datentransfers $\mathbf{t}$. Die Endzeiten z_E^j und z_E^t können durch Addition der Gesamtzeit zur Startzeit der jeweiligen Aktion berechnet werden.

Zu beachten ist, dass ein Job erst ausgeführt werden kann, wenn der letzte Datentransfer abgeschlossen ist:

- $z_E^{max}(T^L) = \max(z_E^t)$ sei der Endzeitpunkt des letzen Datentransfers, wobei T^L die Menge aller für den Job notwendigen Datentransfers entsprechend (7b) ist.
- $z_E^{max} \leq z_S^j$ (13)

Die Startzeit des gesamten Auftrags ist gleich der frühesten Startzeit eines Datentransfers.

- $z_S(T^L) = \min(\{z_S^t \mid t \in T^L\})$ (14)

Die Endzeit des gesamten Auftrags ist gleich der Endzeit des eigentlichen Jobs.

- $z_E(j, r) = z_E^j = z_S^j + a^j / at^r$ (15)

Aus (14) und (15) ergibt sich die Gesamtausführungszeit für einen Auftrag $\langle a_j, A^j, D^j \rangle$ und eine mögliche Lösung $\langle r, S^* \rangle$:

- $tt(j, r, S^*) = z_E(j, r) - z_S(T^L) = z_S^j + a^j / at^r - z_S(T^L)$ unter der Nebenbedingung (13).

T^{L^*} ergibt sich aus r und S^*.

Die Belegungen für r und S^* müssen nun so gewählt werden, dass die Gesamtzeit minimiert wird. Dies geschieht entsprechend der Minimierung der Gesamtkosten im vorherigen Abschnitt. Es kann keine andere Belegung $\langle r', S' \rangle$ geben, deren Gesamtzeit geringer ist als die der Belegung $\langle r, S^* \rangle$:

- $S' = \left\{ s \middle| s \in D \wedge \forall d \in D^j : \exists s' \in S' : d \in D^{s'} \wedge \langle s', r \rangle \in N \wedge |S'| = |D^j| \right\}$
- $L^* = \left\{ \langle r, S^* \rangle \middle| \forall r' \in R, r' \neq r, \forall S' \neq S^* : tt(j, r, S^*) \leq tt(j, r, S') \right\}$ (16)

Entsprechend Formel (12) muss anschließend noch die beste Lösung aus der Menge der verbliebenen Lösungen L^* gewählt werden.

3.2.3 Klassifizierung des Optimierungsproblems

Soll der Resourcebroker nur eine passende Ressource und Datenquellen für die benötigten Dateien finden, so wäre (7) eine ausreichende Lösung. In diesem Falle gehört das Modell in die Klasse der *Constraint-Satisfaction Problems* (CSP) [25]. Ein CSP umfasst eine Menge von Variablen, jede mit einer endlichen und diskreten Wertedomäne, und eine Menge von Bedingungen bzw. Einschränkungen. Durch diese Bedingungen wird die Menge aller möglichen Variablenbelegungen durch Werte der jeweiligen Domänen eingeschränkt. Ziel des CSP ist es, eine

solche Variablenbelegung zu finden, die alle Bedingungen erfüllt. Daher ist das CSP nicht ausreichend, weil das Lösungsverfahren abbricht, sobald die erste Lösung gefunden wurde. Im Modell müssen aber zunächst alle möglichen Lösungen gefunden werden, um anschließend die beste aller gefundenen Lösungen auszuwählen.

Vielmehr lässt sich das Problem in die Klasse des *Integer Programming* [42] einordnen, einer speziellen Form des *Linear Programming* wobei alle Variablen nur ganzzahlige Werte annehmen können. Vorteil dieser Problemklasse ist, dass beispielsweise logische Werte modelliert werden können, indem der Wertebereich einer Variablen auf [0,1] begrenzt wird. Dies verhindert unnatürliche Ergebnisse. So ist es nicht sinnvoll, „0,6 Ressourcen“ zu benutzen, weil diese tatsächlich nicht zur Verfügung stehen[10]. Alle Variablen im Modell können nur ganzzahlige Werte annehmen. In einigen Formeln wird die Division benutzt, welche nicht garantiert ein ganzzahliges Ergebnis zurückgibt. In diesem Fall wird das Ergebnis aufgerundet.

[10] Eine Ausnahme wäre, wenn man davon ausginge, dass ein Job aufgeteilt werden kann, wobei durch die parallele Ausführung die Gesamtzeit gesenkt werden könnte. Auch könnte eine Datei von mehreren Replikaten gleichzeitig gelesen werden, um die Übertragungszeiten zu verringern. Diese Fälle werden im Modell nicht betrachtet.

4 Ein Brokeransatz: ClassAds und Matchmaking

Nachdem im vorherigen Kapitel das Modell des Resourcebrokers beschrieben wurde, sollen in diesem und den beiden folgenden Kapiteln heute verfügbare Systeme vorgestellt werden, auf denen der in dieser Arbeit entwickelte Resourcebroker aufbaut bzw. mit denen er zusammenarbeitet. Zunächst wird eine Beschreibungssprache für Aufträge und Ressourcen im Grid sowie das Condor-System, für welches die Sprache ursprünglich entwickelt wurde, präsentiert. Außerdem wird beschrieben, wie für einen Auftrag passende Ressourcen ausgewählt werden. Dieser Mechanismus wird im Kontext von Condor „Matchmaking“ genannt.

4.1 Condor

Das Condor *High Throughput Computing System* [9] ist ein Workload Management System, entwickelt vom Condor Research Project[11] der University of Wisconsin-Madison, USA. Es ermöglicht das Planen sowie das Ausführen von rechenintensiven Jobs und bietet dafür viele benötigte Funktionen an, u.a. Zeitplanung für Jobs sowie Verwaltung und Überwachung von Ressourcen.

Eine Besonderheit von Condor ist, dass es ungenutzte Rechenkapazitäten von Desktoprechnern benutzen kann. Dafür wird auf entsprechenden Rechnern ein Dienst gestartet, der das Tastatur- und Mausverhalten überwacht. Wird für eine festgelegte Zeitspanne keine Tastatur- und Mausaktivität festgestellt, beginnt Condor mit der Ausführung von anstehenden Aufträgen (so genannten *Jobs*). Stellt Condor fest, dass der Rechner nicht mehr zu Verfügung steht, wird der laufende Job angehalten. Durch einen speziellen Checkpoint-Mechanismus ist es Condor möglich, den Job zu einem späteren Zeitpunkt fortzusetzen.

Condor bietet außerdem Unterstützung für serielle und parallele (MPI, PVM) Jobs in Clustern an und kann auch in Gridumgebungen zusammen mit Globus benutzt werden. Dazu existiert Condor-G, eine Erweiterung von Condor, die eine Nutzung von Globus über die Standardschnittstellen von Condor ermöglicht.

Zur Ausführung von Jobs muss zunächst eine Umgebung eingerichtet werden – der *Condor-Pool*. Ein solcher Pool besteht aus einem zentralen Manager (*central manager*) und einer beliebigen Anzahl anderer Rechner. Der Manager sammelt permanent Informationen über den Zustand des Pools und ist für die Zuteilung von Ressourcen zu Jobanfragen, das so genannte Matchmaking (siehe Abschnit 4.3), zuständig. Konzeptionell betrachtet ist ein Pool eine Menge von Ressourcen (Rechnern, in Condor auch *Machines*) und Ressourcenanfragen (Jobs). Jede Ressource des Pools teilt in periodischen Abständen dem zentralen

[11] http://www.cs.wisc.edu/condor/

Manager seinen aktuellen Zustand mit. Dazu gehören statische Angaben wie z.B. die Größe des Speichers oder die CPU-Leistung und dynamische wie z.B. die aktuelle Speicher- oder CPU-Auslastung. Außerdem können für die Jobs und Rechner Bedingungen festgelegt werden, die bei einer Jobausführung zu beachten sind. So kann für einen Job formuliert werden, dass seine Ausführung ein bestimmtes Betriebssystem auf der auszuwählenden Ressource benötigt. Umgekehrt kann für einen Rechner bestimmt werden, dass dieser z.B. nur Jobs bearbeitet, die von ausgewählten Benutzern abgeschickt wurden. Sämtliche Informationen und Bedingungen werden mit Hilfe von *ClassAds* formuliert, welche im Folgenden vorgestellt werden.

4.2 Aufbau von ClassAds

Classified Advertisements („geordnete Anzeigen“, kurz *ClassAds*) ermöglichen eine flexible Beschreibung von Eigenschaften und Anforderungen von Ressourcen. Im Condor-System dienen sie zur Beschreibung von Ressourcen und Jobs. Sie bestehen aus einer Menge eindeutig benannter Ausdrücke (*expressions*). Jeder benannte Ausdruck wird als Attribut (*attribute*) bezeichnet und setzt sich aus einem Attributnamen und einem Attributwert zusammen.

Inzwischen gibt es zwei Versionen der ClassAd-Sprache. Die erste Version ist in ihrer Funktionalität beschränkt, wird aber bis heute[12] für Condor verwendet, da sich die Implementation der neuen Version kompliziert gestaltet. Neue Dienste des Condor-Teams wie z.B. Stork[13], ein Scheduler für Datenmanagementaufgaben, benutzen heute schon die neue ClassAd-Version.

Im Folgenden werden beide Versionen ausführlich beschrieben. Zunächst war geplant, Condor so zu erweitern, dass es neben der Wahl eines Rechners zur Ausführung eines Jobs auch auf Informationen von Datenmanagementsystemen zugreifen kann. Auf Grund der Beschränktheit der alten ClassAd-Version war dies aber nicht praktikabel realisierbar. So bietet die alte Version keine verschachtelten ClassAds. Damit ist es kompliziert, die Anforderungen eines Jobs an eine Ressource von den Anforderungen an einen Datenspeicher zu trennen. Darüber hinaus bietet die alte Version kein API an, wodurch der nötige Programmieraufwand steigt. In Kapitel 8 werden die Nachteile der alten Sprachversion ausführlicher besprochen. Obwohl nur die neue Version der Sprache in dieser Arbeit benutzt wird, soll trotzdem zunächst die alte Version vorgestellt werden, da diese bei Condor noch zum Einsatz kommt und der neuen Version als Basis dient. Anschließend werden die Erweiterungen der neuen Version vorgestellt.

[12] siehe auch [13]
[13] http://www.cs.wisc.edu/condor/stork

4.2.1 ClassAds im Condor System

Für die alten ClassAds ist keine komplette Sprachreferenz erhältlich. Es findet sich aber im Condor-Handbuch [13] eine umfassende Syntaxbeschreibung.

Ein Ausdruck einer ClassAd kann entweder ein Literal, eine Attributreferenz oder eine Kombination solcher mittels Operatoren sein. Ein Literal kann hierbei ein Integer, ein Real oder ein String sein. Die Schlüsselwörter **TRUE** und **FALSE** entsprechen den Werten 1 bzw. 0. Darüber hinaus gibt es Typen für „undefiniert“ und „Fehler“, repräsentiert durch die Schlüsselwörter **UNDEFINED** und **ERROR**. Abb. 5 (links) zeigt Beispiele für einfache Ausdrücke.

```
[
  Memory = 128;
  LoadAvg = 0.33;
  Machine = "maggie.zib.de";
]
```

```
[
  a = 1;
  b = a;
  c = b;
]
```

Abb. 5 Beispiele für Ausdrücke (links) und Referenzen (rechts) in einer ClassAd
In diesem Beispiel referenziert „b“ das Attribut „a“ und „c“ das Attribut „b“.

ClassAds unterliegen keinem festen Schema, wodurch sie extrem flexibel und auch zur Beschreibung von anderen Objekten als Jobs und Ressourcen geeignet sind. Im Zusammenhang mit Referenzen ist dieses Prinzip aber fehlerbehaftet. Um dies zu verstehen, muss an dieser Stelle kurz das Prinzip des Brokers vorgestellt werden. Bei Condor werden immer genau zwei ClassAds miteinander *gematcht*. Dabei wird nach Auswertung bestimmter Attribute entschieden, ob zwei ClassAds zueinander passen oder nicht. Einfache Datentypen werden immer zu sich selbst ausgewertet. Referenzen ermöglichen es, sowohl auf Attribute in der eigenen als auch in der zu vergleichenden ClassAd zuzugreifen (siehe Abb. 5 rechts). Wird ein Attribut mit einer Referenz ausgewertet, so wird zunächst das Ziel der Referenz gesucht. Deren Wert ist dann auch gleichzeitig der ausgewertete Wert des Attributs. Da eine zu vergleichende ClassAd zum Zeitpunkt der Erstellung einer ClassAd unbekannt ist, gibt es keine Möglichkeit zu garantieren, dass eine Attributreferenz tatsächlich existiert. Wird keine äquivalente Attributreferenz gefunden, so wird der Wert des Attributs, das die Referenz enthält, auf **UNDEFINED** gesetzt. Um Überschneidungen im Namensraum zu verhindern, können bei Referenzen die Präfixe **MY.** und **TARGET.** benutzt werden, um anzugeben, dass ein Attribut in der eigenen bzw. in der anderen ClassAd referenziert wird. Ist kein Präfix vorhanden, wird zunächst in der eigenen ClassAd gesucht. Ist dort die Suche erfolglos, wird in der zu matchenden ClassAd gesucht und anschließend in der ClassAd-Umgebung[14]. Konnte auch dort kein Attribut mit dem gesuchten Bezeichner gefunden werden, wird **UNDEFINED** zurückgegeben. Wurde

[14] Eine Menge von Ausdrücken, die für den gesamten Pool gelten.

ein Präfix angegeben, so wird nur in der jeweiligen ClassAd gesucht. Ist das Attribut dort nicht vorhanden, wird ebenfalls **UNDEFINED** zurückgegeben.

Für den Fall, dass während der Auswertung auf ein Attribut zurückgegriffen wird, dass sich selbst gerade in der Auswertung befindet, so handelt es sich um eine zirkuläre Abhängigkeit, welche nicht zulässig ist. In diesem Fall wird **ERROR** zurückgegeben.

Attribute besitzen keinen festgelegten Typ. Daher ist eine Typüberprüfung erst im Moment der Auswertung möglich. Werden für einen Operator falsche Typen benutzt, so ist das Ergebnis des Operators **ERROR** bzw. **UNDEFINED**, d.h. die Operatoren sind total[15] für alle möglichen Operanden. Beispielsweise liefert die Multiplikation einer Ganzzahl mit einer Zeichenkette (**3 * "fünf"**) den Wert **ERROR**.

Hohe Präzedenz	**- (unäre Negierung)**
↓	*** /**
	+ -
	< <= >= >
	== != =?= =!=
	&&
Geringe Präzedenz	**||**

Tab. 1 Operatoren der ClassAd-Sprache, [13]

Die arithmetischen Operatoren **+**, **-**, ***** und **/** arbeiten auf Ganzzahlen (*Integers*) und Gleitkommazahlen (*Reals*). Gegebenfalls werden Ganz- in Gleitkommazahlen gewandelt, wenn ein Operand eine Gleitkomma- und einer eine Ganzzahl ist. Für andere Typen ist das Ergebnis **ERROR**. Die Präzedenz der einzelnen Operatoren zeigt Tab. 1.

Die vergleichenden Operatoren **==** (Gleichheit), **!=** (Ungleichheit), **<, >, <=, >=** arbeiten auf Ganz- und Gleitkommazahlen sowie Zeichenketten (*Strings*). Ganzzahlen werden beim Vergleich mit Gleitkommazahlen automatisch in Gleitkommazahlen gewandelt. Ein Vergleich zwischen Zeichenketten und Zahlen ist nicht möglich und hat **ERROR** als Ergebnis.

Operand 1	**Operand 2**	**Operation**			
		==	**=?=**	**!=**	**=!=**
10	10	TRUE	TRUE	FALSE	FALSE
10.0	10	TRUE	FALSE	FALSE	TRUE
10	UNDEFINED	UNDEFINED	FALSE	UNDEFINED	TRUE
10	ERROR	FALSE	FALSE	ERROR	TRUE

Tab. 2 Beispiel für Vergleichsoperationen

Die Operatoren **=?=** und **=!=** verhalten sich ähnlich wie **==** und **!=**, nur verlangt **=?=** dass beide Operanden nicht nur den gleichen Wert haben, sondern auch vom gleichen Typ sind. **=!=** kann als Test für „ist nicht identisch mit“ benutzt werden.

[15] D.h., es gibt wohl definierte Werte für alle möglichen Operatoren.

=!= und **=?=** haben als Ergebniswert immer **true** oder **false**. Beispiele für diese Operatoren finden sich in Tab. 2.

Die logischen Operatoren **&&** (Und) und **||** (Oder) arbeiten auf allen Zahlentypen, wobei 0 als falsch und alle anderen Werte als wahr interpretiert werden. Im Gegensatz zu den anderen Operatoren (außer **=?=** und **=!=**) sind die logischen Operatoren nicht strikt. D.h. sie können, auch wenn ein Operand **UNDEFINED** oder **ERROR** ist, ein anderes Ergebnis zurückliefern (siehe auch Tab. 3).

&&	T	F	U	E
T	T	F	U	E
F	F	F	F	F
U	U	F	U	E
E	E	E	E	E

||	T	F	U	E
T	T	T	T	T
F	T	F	U	E
U	T	U	U	E
E	E	E	E	E

Tab. 3 Wahrheitstabellen für die Und- und die Oder-Operation

Für das Matchmaking, also das Suchen passender ClassAds, muss jede ClassAd zwingend zwei Attribute besitzen: **MyType** und **TargetType** mit den festgelegten Werten „Job“ und „Machine“, wenn es sich um eine Job-ClassAd handelt bzw. mit „Machine“ und „Job“, wenn es sich um eine Machine-ClassAd handelt.

4.2.2 Neue ClassAds

Die neue Version [11, 13, 43] stellt eine Erweiterung der vorherigen Version da. Zu den Neuerungen gehört, dass es mehr Datentypen gibt (Boolesche Werte, Zeitstempel und Zeitspannen) und dass Ausdrücke auch Wertelisten enthalten können. Diese werden durch geschweifte Klammern repräsentiert. Der Zugriff auf einzelne Werte erfolgt durch Angabe des Indexes (beginnend bei 0 für das erste Element) in eckigen Klammern (siehe Abb. 6).

```
[
  a = {5, "foo", true};
  b = a[0];
]
```

Abb. 6 Listendefinition und Zugriff auf Liste in ClassAds
Der Ausdruck a ist eine Liste mit drei Elementen. Da die Elemente selbst Ausdrücke sind, können sie von unterschiedlichem Typ sein. Der Ausdruck b verweist auf das erste Element von a, d.h. bei der Auswertung ergibt sich für b der Wert 5.

Eine weitere Neuerung der neuen Sprachversion ist, dass ClassAds selbst Ausdrücke sind. Somit wird es möglich, ClassAds zu schachteln, wobei die äußerste ClassAd als *root scope* bezeichnet wird, also als äußerster Gültigkeitsbereich. Jede geschachtelte ClassAd besitzt einen sie umgebenden Gültigkeitsbereich (*parent scope*), der identisch mit der ClassAd ist, in der die geschachtelte ClassAd

erzeugt wurde. Abb. 7 zeigt ein Beispiel für die Verschachtelung. Um auf die äußere ClassAd zugreifen zu können, gibt es den reservierten Ausdruck **parent**. ClassAds sind selbstauswertend, d.h. der Wert einer ClassAd ist die ClassAd selbst.

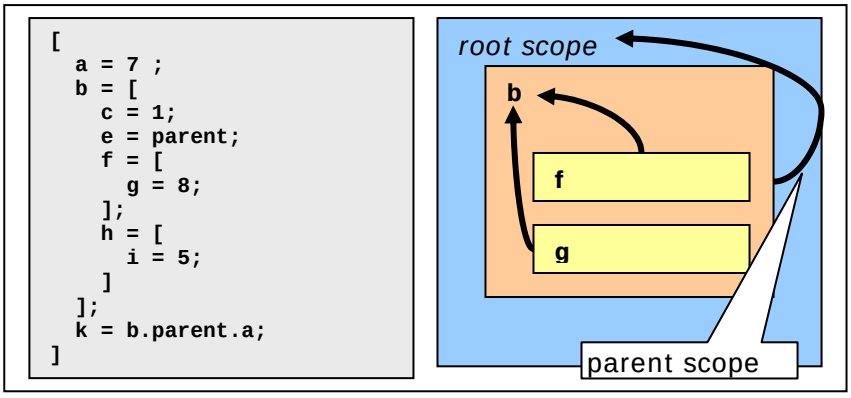


Abb. 7 Gültigkeitsbereiche und Verschachtelung von ClassAds
Die Ausdrücke b, f und g sind geschachtelte ClassAds, wobei f und g doppelt geschachtelt sind. Der rechte Teil der Abbildung verdeutlicht dies grafisch. k greift auf den *parent scope* von b zu, welcher der *root scope* selbst ist. Das im *root scope* vorhandende Attribut a hätte auch durch k=a abgefragt werden können.

Auf Grund der Schachtelung ändert sich die Auswertung von Attributreferenzen. Man unterscheidet zwischen reinen Attributnamen (**attr**), Attributnamen mit vorangestelltem Punkt (**.attr**) und Attributnamen, die zusammen mit einem Ausdruck angegeben werden (**expr.attr**). Im ersten Fall wird in der aktuellen ClassAd nach einem Attribut mit dem Namen gesucht, wird dort kein Attribut mit diesem Namen gefunden, wird im darüber liegenden Gültigkeitsbereich gesucht usw. Wird auch in den äußersten Gültigkeitsbereichen (*root scopes*) der eigenen und der matchenden ClassAd kein passendes Attribut gefunden, so ist das Ergebnis **UNDEFINED**. Im Falle des vorangestellten Punkts wird nur im *root scope* gesucht. Im letzten Fall wird zunächst versucht, den Ausdruck **expr** aufzulösen (nach dem gleichen Prinzip wie im ersten Fall), wobei **expr** eine ClassAd sein muss. Anschließend wird innerhalb der ClassAd nach dem Attribut gesucht.

Die Sprache stellt drei reservierte Bezeichner für fest definierte Referenzen bereit (siehe auch Abb. 8): **self** zum Zugriff auf die ClassAd, die das aktuelle Attribut enthält, **parent** für den Zugriff auf die darüberliegende ClassAd und **root** als Referenz auf die äußerste ClassAd (*root scope*). Ein Aufruf von **parent** in der äußersten ClassAd resultiert bei der Auswertung in dem Wert **UNDEFINED**.

Darüber hinaus bietet die neue Version eine Menge von Funktionen der Form **name(arg0, …, argn)** um z.B. den Typ von Attributen abzufragen, um Zeichenkettenmanipulationen durchzuführen oder um Zeit- und Datumswerte abzufra-

gen. Durch kleine Änderungen an den Bibliotheken der APIs können auch selbstdefinierte Funktionen[16] eingebunden werden. Diese ermöglichen es, während des Matchmakings dynamische Informationen abzurufen.

Aktuell gibt es zwei Implementierungen der neuen ClassAd-Sprache für C++ und Java. Beide Versionen sollen nach außen das gleiche Verhalten zeigen. Zurzeit gibt es aber noch Unterschiede. So hat die Java-Version das Problem, mehrere **parent**-Aufrufe hintereinander (**c = parent.parent.a**) nicht ausführen zu können. Stattdessen wird immer **UNDEFINED** zurückgeliefert.[17]

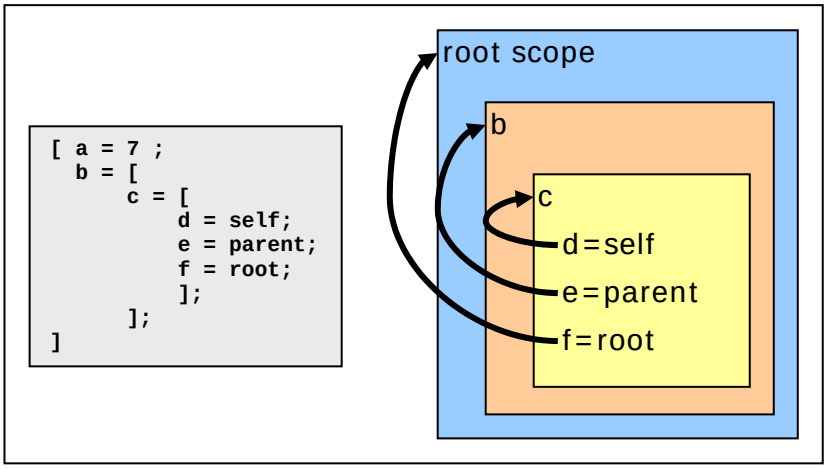


Abb. 8 Verwendung der Schlüsselwörter *self*, *parent* und *root*

4.3 Matchmaking

Die Hauptaufgabe des Matchmakers bzw. des zentralen Managers ist das Matchen [31] von Ressourcen und Anfragen, d.h. das Zusammenführen von verfügbaren Rechnern mit passenden Jobs. Dies ist vergleichbar mit dem Anzeigenteil einer Zeitung, wo Anbieter eine Ware anpreisen und Kunden bestimmte Waren suchen. Dabei hat jede Seite ihre speziellen Anforderungen (z.B. Mindestpreis und maximale Zahlungsbereitschaft). Bei Condor sind die Anbieter die Inhaber von Ressourcen und die Käufer die Nutzer, die Rechenzeit benötigen, um ihre Jobs auszuführen. Computerressourcen sind die Ware. Was angeboten und nachgefragt wird, wird durch die ClassAds beschrieben.

Findet ein Kunde eine Anzeige interessant, so muss er sich an den Anbieter wenden und beide müssen einen Vertrag aushandeln. Bei Condor hingegen werden alle ClassAds permanent durch eine zentrale Instanz miteinander verglichen. Findet diese eine verfügbare, zum Job passende Ressource, so werden die beiden

[16] In Condor ist es grundsätzlich möglich, benutzerdefinierte Funktionen zu implementieren. Diese Methode ist aber jedoch nicht offiziell dokumentiert.

[17] Bis zum Zeitpunkt der Fertigstellung dieser Arbeit konnte das Condor-Team keine Lösung für dieses Problem bereitstellen.

ClassAds gematcht und Condor ist für die Ausführung des Jobs verantwortlich. Wird mehr als eine Ressource gefunden, wird diejenige benutzt, die der Nutzer präferiert. Dies hat gegenüber dem Anzeigenbeispiel den Vorteil, dass sich weder Kunde noch Anbieter kennen müssen. Auch wird dem Kunden die Arbeit abgenommen, den besten der verfügbaren Anbieter auszuwählen. Nachteil an diesem Verfahren ist aber, dass beide Seiten alle Anforderungen eindeutig in einer ClassAd festlegen müssen. Abb. 9 fasst die Schritte des Matchmakings zusammen.

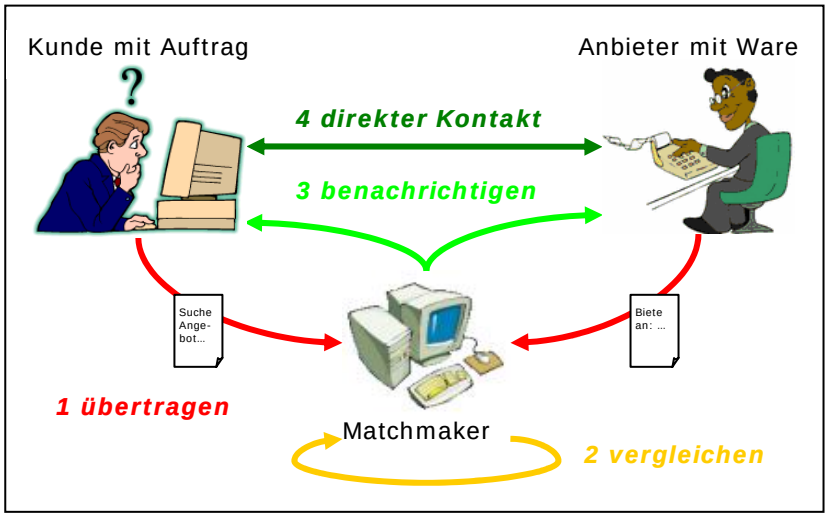


Abb. 9 Überblick über das Matchmaking, nach [31]
(1) Anbieter und Nachfrager übertragen eine ClassAd mit den notwendigen Informationen an den Matchmaker. (2) Der Matchmaker vergleicht die ClassAds miteinander. (3) Findet er zueinander passende ClassAds, werden deren Besitzer informiert. (4) Anschließend können beide miteinander kommunizieren.

Das Prinzip des Matchmakings ist nicht auf Jobs und Rechner begrenzt. Die Sprache erlaubt es, beliebige ClassAds miteinander zu vergleichen und zu matchen. Bei geschachtelten ClassAds ist es möglich, nur eine der inneren ClassAds mit einer anderen ClassAd zu matchen. Somit ist man nicht mehr auf Vergleiche zweier ClassAds (*bilaterales Matchmaking*) angewiesen. In diesem Fall spricht man vom *multilateralen Matchmaking*.

Das Matchmaking selbst umfasst mehrere Schritte. Zunächst werden die beiden ClassAds in einer neuen ClassAd zusammengefasst, wie es Abb. 10 zeigt. **`Ad1.self`** enthält die eine der beiden bisherigen ClassAds, **`Ad2.self`** die andere. Durch die Referenz des **`Other`**-Attributs kann auf die andere ClassAd verwiesen werden. Sofern in den Original-ClassAds kein Attribut **`other`** existiert, wird – wie oben beschrieben – automatisch im darüberliegenden Gültigkeitsbereich gesucht. In einem zweiten Schritt werden die **`Requirements`**-Attribute **`Ad1.self.Requirements`** und **`Ad2.self.Requirements`** ausgewertet.

```
[
  Ad1 = [
    self = [ Original-ClassAd 1];
    other = Ad2.self;
  ]
  Ad2 = [
    self = [ Original-ClassAd 2];
    other = Ad1.self;
  ]
]
```

Abb. 10 Beispiel für eine MatchedClassAd

Das **REQUIREMENTS**-Attribut ist ein Ausdruck, der zu einem der Wahrheitswerte „wahr“ oder „falsch“ ausgewertet wird. Ergibt die Auswertung des Attributs den Wert **UNDEFINED**, so wird der Wahrheitswert „falsch“ anstelle dessen benutzt. In dem Ausdruck können Anforderungen formuliert werden, die die Attribute der eigenen oder die der anderen ClassAd erfüllen müssen. Abb. 11 zeigt ein Beispiel für eine JobClassAd. Das **REQUIREMENTS**-Attribut verlangt hier von der matchenden ClassAd (durch das **other**-Schlüsselwort benannt), dass sie vom Typ „Machine“ ist, d.h. es muss in der anderen ClassAd ein Attribut mit dem Namen „Type“ und dem Wert „Machine“ geben. Außerdem muss es noch zwei weitere Attribute („Arch“ und „OpSys“) mit den entsprechenden Werten geben.

```
[
  Type = "Job"
  Requirements = other.Type=="Machine"
    && other.Arch=="INTEL" && other.OpSys="LINUX"
  Rank = other.Memory
  Cmd = "/home/test-exe"
  Department = "CompSci"
  Owner = "user1"
  DiskUsage = 6000
]
```

Abb. 11 Beispiel für eine JobClassAd
Für den Match sind nur die Attribute *Requirements* und *Rank* von Bedeutung. Andere Attribute können durch den Benutzer selbst festgelegt und ggf. durch *Requirements* und *Rank* referenziert werden. *Type* gibt hier den Typ der ClassAd an, *Cmd* das auszuführende Programm, *Department* den Namen der Organisation, zu der der Auftraggeber gehört, *Owner* den Auftraggeber und *DiskUsage* den benötigten Speicherplatz.

Für den Fall, dass es mehr als eine passende ClassAd gibt, werden zusätzlich die **Rank**-Attribute ausgewertet. Der Verfasser einer ClassAd ist dafür verantwortlich, dass der Wert des **Rank**-Attributs immer zu einem Real ausgewertet werden kann. Zur Ausführung kommt schließlich die ClassAd-Paarung mit dem höchsten Rang. Gibt es mehrere ClassAd-Paarungen mit dem gleichen Rang, wird die zuerst gefundene Paarung benutzt. Die JobClassAd in Abb. 11 beispielsweise, bevorzugt einen Rechner mit möglichst viel Arbeitsspeicher („Memory“).

Abb. 12 zeigt eine weitere ClassAd vom Typ „Machine“. Auch diese ClassAd besitzt ein **REQUIREMENTS**- und ein **RANK**-Attribut. Die ClassAd verlangt bei einem

Match, dass die Auslastung der eigenen Ressource geringer als 30% ist. Es mag verwundern, warum sich die Ressource überhaupt anbieten sollte, wenn ihre eigene Auslastung größer als 30% ist. Der Grund hierfür ist, dass bestimmte Attribute automatisch durch Condor eingetragen werden, hier beispielsweise die aktuelle Auslastung, und der Besitzer der Ressource keine Kontrolle über die Erstellung der ClassAds hat. Er kann nur von vornherein festlegen, unter welchen Bedingungen die ClassAd matchen soll. Des Weiteren verlangt die ClassAd, dass die andere ClassAd vom Typ „Job“ ist und der Organisation „CompSci“ angehört. Das **Rank**-Attribut bevorzugt Aufträge, die möglichst wenig Festplattenspeicher benötigen.

```
[
  Type = "Machine"
  Machine = "nostos.wisc.edu";
  Requirements = LoadAvg<=0.3 && other.Deparment=="CompSci"
    && other.Type=="Job"
  Rank = 1/other.DiskUsage
  Arch = "INTEL"
  OpSys = "LINUX"
  Disk = 3076076
  LoadAvg = 0.25
]
```

Abb. 12 Beispiel für eine MachineClassAd
Machine gibt den Namen des Rechners an. *Arch* gibt dessen Architektur und *OpSys* dessen Betriebssystem an. *Disk* bezeichnet die freie Festplattenkapazität.

Es ist leicht ersichtlich, dass die **REQUIREMENTS**-Attribute beider ClassAds wahr ergeben, wenn man sie miteinander matcht. Sollte es weitere ClassAds geben, die mit einer der beiden ClassAds matchen, so müssen deren **RANK**-Attribute verglichen werden. Zu beachten ist dabei, dass jede beteiligte ClassAd einen eigenen Rangwert hat. Angenommen, es gibt drei ClassAds A, B und C mit den in Tab. 4 gezeigten Rangwerten.

Match A-B	A.Rank = 10	B.Rank = 5
Match A-C	A.Rank = 6	C.Rank = 11
Match B-C	B.Rank = 10	C.Rank = 7

Tab. 4 Rank-Werte verschiedener Matches

Hier bevorzugt ClassAd A die ClassAd B, B bevorzugt C und C bevorzugt A. Es gibt also keine „beste“ Kombination. Vielmehr muss durch den Matchmaker eine Entscheidung getroffen werden. Wie dies bei Condor realisiert wurde, ist den Ausführungen einschlägiger Literatur nicht zu entnehmen. Der in dieser Arbeit vorgestellte Resourcebroker betrachtet z.B. zunächst nur den Rank-Wert der JobClassAd. Nur für den Fall, dass es hier zwei oder mehr ClassAds mit dem gleichen Wert geben sollte, werden weitere Rangwerte betrachtet. Sollte es anschließend noch mehr als eine Kombination mit identischen Rangwerten geben, so wird die zuerst gefundene benutzt.

Weiter oben wurde angesprochen, dass in der neuen ClassAd-Version geschachtelte ClassAds möglich sind und dass man auch nur Teile von ClassAds miteinander matchen kann. Diese Art des Matchmakings ist nicht Teil von Condor, sondern muss auf Basis des ClassAd-APIs selbst implementiert werden. Abb. 13 zeigt auf der linken Seite eine geschachtelte ClassAd. Auf der rechten Seite sind zwei weitere ClassAds abgebildet. Um diese ClassAds miteinander zu matchen, muss jede geschachtelte ClassAd einzeln gematcht werden. Es ist ersichtlich, dass die ClassAd vom Type „C“ zum Ausdruck A passt und die ClassAd vom Type „D“ zum Ausdruck B. Zu beachten ist, dass die Anzahl der möglichen Matchkombinationen exponentiell mit der Anzahl der geschachtelten ClassAds ansteigt. Da die Auswahl der besten Kombination auf Basis der Rank-Attribute erfolgt, müssen zunächst alle möglichen Kombinationen probiert werden.

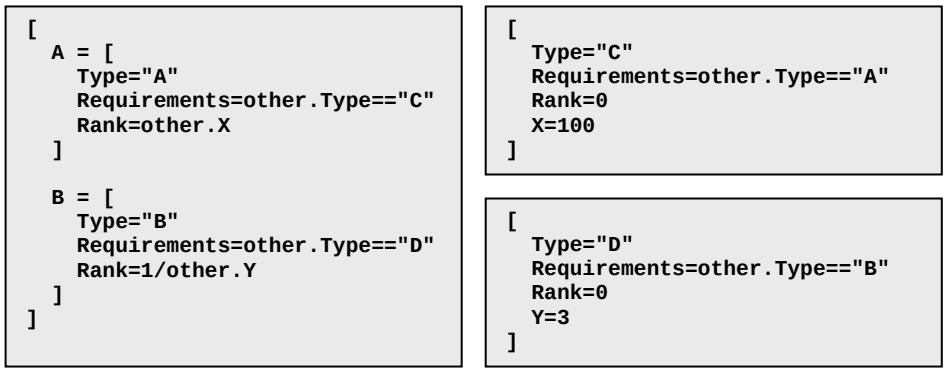


Abb. 13 Drei ClassAds, die linke ClassAd ist geschachtelt

5 Ein Grid-Datenmanagementsystem: ZIBDMS

Das vorherige Kapitel hat ein Beispiel aufgezeigt, wie Ressourcen, Jobs und Daten durch ClassAds beschrieben werden können und wie durch Matchmaking zueinander passende Komponenten gefunden werden können. Dieses Kapitel beschäftigt sich damit, wie Daten gespeichert werden und wie der Zugriff auf diese erfolgen kann. Es wird beschrieben, wie ein *Datenmanagementsystem* (DMS) aufgebaut ist und wie der Zugriff auf Daten organisiert wird – zunächst allgemein und anschließend an einer konkreten Implementierung, dem ZIBDMS.

5.1 Definition eines DMS

Ein *Datenmanagementsystem* dient der Verwaltung und dem Zugriff auf verteilt vorliegende Daten. Ein *Grid-Datenmanagementsystem* beachtet darüber hinaus die Besonderheiten eines Grid-Systems, etwa die Heterogenität der einzelnen Ressourcen sowie die Größe der gesamten Systemumgebung [20].

Durch die Vielzahl der Ressourcen eines Grid ist es für den Anwender kaum möglich, den physikalischen Speicherort einer benötigten Datei zu kennen. Darüber hinaus unterscheiden sich auch verschiedene Datenspeicher durch die zur Verfügung gestellten Schnittstellen. So muss der Anwender nicht nur den Speicherort, sondern auch die Schnittstelle bzw. ein bestimmtes Protokoll kennen und benutzen, um Zugriff auf die Daten zu erhalten. Ein DMS hat hier zum Ziel, eine transparente Schnittstelle zu liefern, die dem Benutzer den einheitlichen Zugriff auf Daten ermöglicht, egal ob diese als Datei, URL[18], Datensatz einer Datenbank oder in einer anderen Form vorliegen.

Zum anderen existieren von einer Datei häufig Replikate auf verschiedenen Datenspeichern. Dies erhöht die Verfügbarkeit einer Datei und ermöglicht eine Ausfalltransparenz, solange noch mindestens ein Replikat erreichbar ist. Greift ein Anwender auf eine Datei lesend zu, so kann gleichzeitig von mehreren Replikaten gelesen werden, um so die Datentransfergeschwindigkeit zu erhöhen. Nachteil bei der Verwendung von Replikaten ist, dass das Bearbeiten von Dateien kompliziert ist, da Änderungen entweder an allen Replikaten gleichzeitig stattfinden müssen oder diese zunächst an einem ausgewählten Replikat durchgeführt und anschließend auf alle anderen übertragen werden müssen[19]. Diese Arbeit geht davon aus, dass ein DMS vornehmlich als Datenlieferant dient, so wie es bei vielen Grid-Systemen üblich ist. Im C3-Grid beispielsweise werden alle Daten, z.B. Satellitenbilder, lokal erstellt und gespeichert. Später wird dann anderen

[18] *Uniform Resource Locator*
[19] Zur Synchronisation von Replikaten siehe z.B. [16]

Forschergruppen der Zugriff auf diese ermöglicht, wobei eine Veränderung der ursprünglichen Daten nicht vorgesehen ist [8].

Die Festlegung, ob eine Datei ein Replikat einer anderen ist, kann nicht durch das System getroffen werden, da es sich hierbei um eine Behauptung des Anwenders handelt [10]. Ein DMS würde daher zwei inhaltlich identische Dateien nicht als Replikate verwalten, sofern es nicht von Anwender eine entsprechende Information erhalten hat. Daher gibt es neben dem physischen Namensraum noch einen logischen Namensraum [47]. In diesem werden Metainformationen über Daten unabhängig von deren physischen Namen gespeichert. Zusätzlich ist eine Abbildung von logischen Namen auf physische Speicherorte notwendig. Es ist grundsätzlich möglich, dass eine Datei mehr als einen logischen Dateinamen (*Logical File Name*, LFN) besitzt. Ein DMS muss also sowohl eine Replikatverwaltung als auch eine Mappingfunktion zwischen logischen und physischen Dateien ermöglichen.

Metadaten bzw. –informationen ermöglichen es, zusätzlich zu den eigentlichen Daten Informationen über diese zu speichern. Dies können automatisch durch das DMS generierte Metadaten sein (beispielsweise die Größe einer Datei) oder durch den Anwender spezifizierte Informationen. Sie erlauben es, eine effiziente Suche auf den Daten zu ermöglichen. Ein großer Vorteil dieser Metadaten ist es, dass diese nicht direkt bei den Replikaten auf den Datenspeichern, sondern zentral beim DMS gespeichert werden können. Damit kann eine Suche erfolgen, ohne dass die einzelnen Datenspeicher kontaktiert werden müssen, was die Suchzeit stark verkürzt.

Neben den hier vorgestellten Funktionen (Erstellung, Suche und Bearbeiten von Datenobjekten[20]) können durch Datenmanagementsysteme oft auch Datentransfers initiiert werden. I.d.R. handelt es sich dabei um *Third-Party-Transfers*, d.h. Daten, die nicht lokal vorhanden sind, werden direkt zwischen zwei Ressourcen ausgetauscht, ohne dass die Bandbreite der Netzwerkverbindung des DMS zur Verfügung gestellt werden muss.

5.2 Das ZIBDMS

Das *ZIB-Datenmanagementsystem* (ZIBDMS) [29, 37, 48] ist ein solches verteiltes System zur Verwaltung von Dateien auf verschiedenen Ressourcen im Grid, welches zurzeit am Zuse-Institut-Berlin entwickelt wird. Es setzt sich aus einer Serverkomponente und verschiedenen Clients zusammen. Im System können mehrere lose gekoppelte Server vorhanden sein. Die Clients greifen auf das System über einen ihnen bekannten Server zu, wie aus Abb. 14 ersichtlich ist.

[20] Datenobjekt dient hier als Oberbegriff für alle möglichen Arten von Daten wie z.B. Dateien, URLs, Datenbanken.

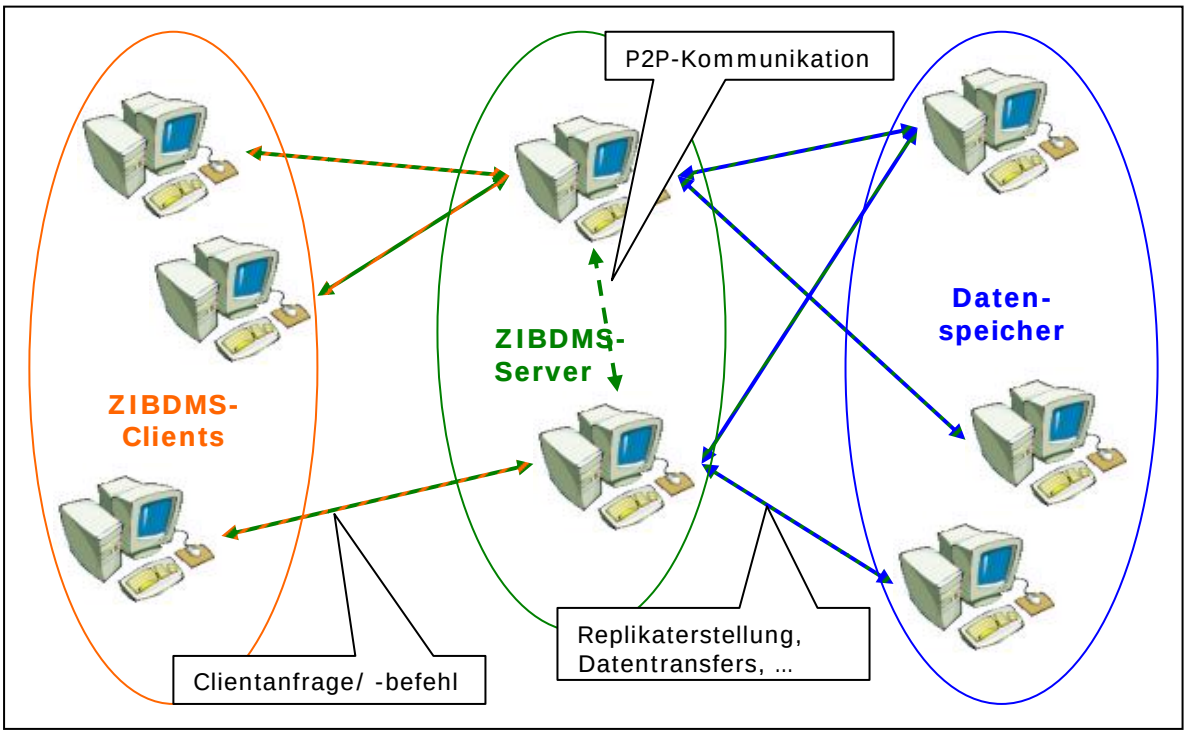


Abb. 14 Kommunikation im ZIBDMS
Ein Client kann immer mit genau einem Server kommunizieren, um Anfragen und Aufträge an das ZIBDMS zu übermitteln. Die Server können durch P2P-Kommunikation Informationen austauschen.

Der Server bietet den Clients verschiedene Sichten auf den *Metadatenkatalog* an und erlaubt darüber hinaus den Zugriff auf die im Katalog referenzierten Objekte. Im Folgenden werden die sowohl bereits implementierten als auch die noch geplanten Funktionen beschrieben. Das ZIBDMS dient in dieser Arbeit als Beispiel für ein Datenmanagementsystem, anhand welchem beschrieben werden soll, wie Resourcebroker Zugriff auf Daten in solchen Systemen erhalten. Darüber hinaus wurde das ZIBDMS im Rahmen dieser Arbeit um einige Funktionen erweitert, die einem Resourcebroker zusätzliche Informationen über geschätzte Transferzeiten von Dateien liefern. Diese Erweiterungen werden im Abschnitt 8.2 ausführlich vorgestellt.

5.2.1 Das ZIBDMS aus Benutzersicht

Zunächst wird die Benutzersicht betrachtet. Das System bietet eine Schnittstelle zu sämtlichen Metadaten des Systems. Der Benutzer hat die Möglichkeit, Suchanfragen an den Metadatenkatalog zu richten und Metadaten zu bearbeiten. Zur Betrachtung der Daten gibt es die hierarchiebasierte und die attributbasierte Sichtweise. Bei der ersten handelt es sich um die klassische Sicht als Verzeichnisbaum. Bei der attributbasierten Sichtweise werden bei Anfragen Attribut-Wert-Paare mit übergeben, woraufhin das System alle Objekte sucht, die die Attribute

mit den entsprechenden Werten besitzen. Es können sowohl Bereichs- als auch Gleichheitsanfragen an das System gerichtet werden.

Der Zugriff auf Daten ist über verschiedene Wege möglich. Zum einen gibt es einen direkten Zugriff, z.B. durch Angabe einer URL, auf eine Objektreferenz. Dadurch kann ein Client direkt auf die Daten zugreifen, ohne dass der Server mit dem Transfer belastet wird. Dies bietet sich an, wenn die Bandbreite zwischen Client und Datenquelle größer ist als die Bandbreite zwischen Datenquelle und Server bzw. Server und Client. Zum anderen kann der Zugriff über eine Schnittstelle des ZIBDMS erfolgen. Hierbei braucht der Client keine Kenntnis darüber zu haben, wo die Daten liegen und mit welchem Protokoll sie abgerufen werden können. Diese Aufgaben bearbeitet der Server und stellt die Daten dann über eine einheitliche Schnittstelle zur Verfügung.

Das ZIBDMS kann zurzeit sowohl auf Daten im lokalen Dateisystem zugreifen als auch auf Daten, die über Standard-URLs erreichbar sind, etwa über die Protokolle HTTP und FTP. Ein Zugriff auf relationale Datenbanksysteme ist denkbar. Als Clients stehen ein Kommandozeilentool, eine Webservice-Schnittstelle, eine graphische Benutzeroberfläche unter Java und ein NFS[21]-Client zur Verfügung.

5.2.2 Objekte im ZIBDMS

Wie oben erwähnt dient der Metadatenkatalog der Verwaltung von Objekten. Ein Objekt ist definiert als eine assoziierte Menge von Attribut-Wert-Paaren. Auf diesen Paaren sind effiziente Suchoperationen möglich. Ein Objekt setzt sich aus verschiedenartigen Attributen zusammen.

Jedes Objekt besitzt ein ***Typattribut***, welches angibt, um welchen Objekttyp es sich handelt. Unterschieden werden *Benutzer-*, *Datei-* und *Ressourcenobjekte*, welche hier kurz vorgestellt werden.

Benutzerobjekte sind geplant, aber noch nicht implementiert. Grids bieten heute schon eine Vielzahl von Authentifizierungsmethoden. Daher soll es im ZIBDMS keine eigene Benutzerverwaltung geben. Vielmehr soll s ein Mapping zwischen extern authentifizierten Benutzern (etwa Windows- oder UNIX-Konten, Globuskennungen) und ZIBDMS-Nutzern und umgekehrt durchgeführt werden. Ein ähnliches Konzept ist für Gruppen angedacht.

Die beschriebene Verteilung der einzelnen Replikate muss im ZIBDMS berücksichtigt werden. Daher setzt sich ein ***Dateiobjekt*** im ZIBDMS aus einem *Unique File Identifier* (UFI), mehreren *Hierarchical File Names* (HFN) und mehreren *Native Storage Locations* (NSL) zusammen. Alle drei Objekttypen besitzen ein Attri-

[21] *Network File System*.

but „UFI“, welches mit einem Primärschlüssel in einer relationalen Datenbank vergleichbar ist (siehe Abb. 15).

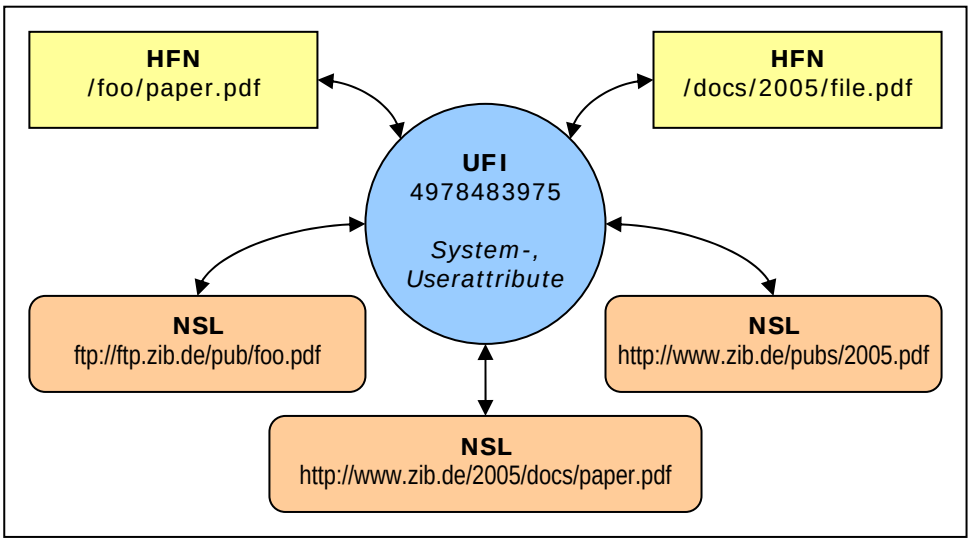


Abb. 15 Zusammenhang von HFN, NSL und UFI
Das Mapping von logischen Dateinamen (HFNs) auf physische Speicherorte (NSLs) erfolgt über einen eindeutigen Bezeichner (UFI).

Für alle Replikate muss das System sicherstellen, dass das UFI-Attribut gleich ist. Inhaltlich identische Datenobjekte mit verschiedenen UFIs werden vom System als unterschiedliche Dateien betrachtet. Die UFI ist zwar ein eindeutiger Bezeichner, für Menschen aber schwer zu erinnern. Daher gibt es, ähnlich wie bei IP-Adressen und URLs, die HFNs, d.h. logische Bezeichner zur Erzeugung hierarchischer Strukturen zwischen den einzelnen Dateien. So wie es zu einer IP-Adresse mehr als eine URL geben kann, sind auch im ZIBDMS mehrere HFNs für eine UFI zulässig. Jeder HFN enthält neben dem eigenen Namen einen Verweis zum übergeordneten HFN. NSLs geben Auskunft über die tatsächlichen Speicherorte der einzelnen Replikate.

Ressourcenobjekte sind zum Speichern von Informationen über Speicherressourcen vorgesehen, zurzeit aber noch nicht implementiert.

Obligatorische Systemattribute sind Attribute, die permanent für ein Objekt vorhanden sein müssen, z.B. die Dateigröße. Diese Attribute werden vom System verwaltet und können durch den Benutzer nicht direkt verändert werden[22].

Optionale Systemattribute*:* Das System verwaltet darüber hinaus noch weitere Attribute, welche jedoch nicht zwingend vorhanden sein müssen. Beispiels-

[22] D.h. der Benutzer kann z.B. die Dateigröße nicht willkürlich ändern. Bearbeitet der Benutzer jedoch den Inhalt der Datei, wodurch sich auch die Dateigröße ändert, so wird das zugehörige Attribut entsprechend angepasst.

weise gehören Attribute dazu, welche festhalten, von wem eine Datei bearbeitet wurde.

***Benutzerattribute**:* Alle anderen Attribute können durch den Benutzer gesetzt und bearbeitet werden. Zu beachten ist, dass Benutzerattribute nicht den Namen eines Systemattributs haben dürfen.

5.2.3 Architektur

Das ZIBDMS ist als ein loser Verbund untereinander kooperierender Server konzipiert. Diese kommunizieren mittels P2P[23]-Protokollen miteinander und sollen so einen verteilten und redundanten Metadatenkatalog verwalten und den Datenaustausch zwischen den einzelnen Servern ermöglichen. Der interne Aufbau des Servers ist in Abb. 16 dargestellt. [48] ordnet die einzelnen Komponenten fünf aufeinander aufbauenden Schichten zu.

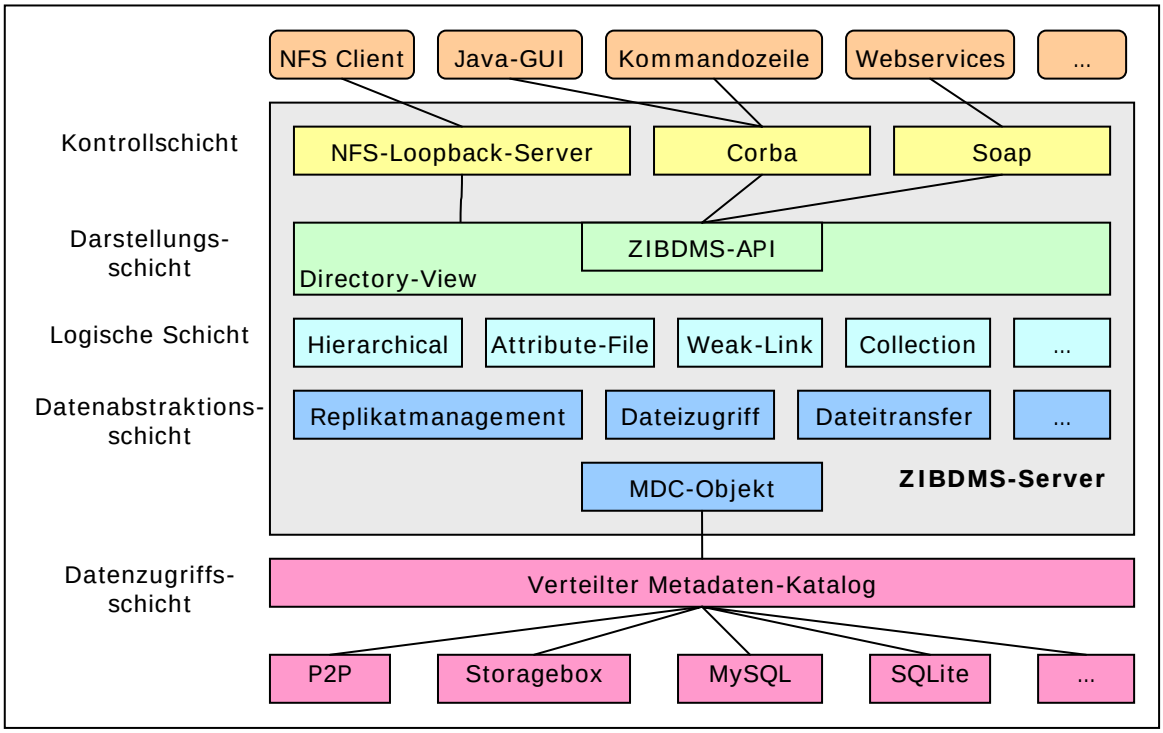


Abb. 16 Softwarearchitektur des ZIBDMS. Nach [29] und [48]

***Datenzugriffsschicht**:* Als unterste Schicht ermöglicht die Datenzugriffsschicht einen direkten Zugang zum (verteilten) Metadatenkatalog und auf Datenobjekte. Der Metadatenkatalog kann in verschiedenen Datenbanken abgelegt werden. Zurzeit werden Storagebox[24], eine am ZIB entwickelte Datenbank mit einem dem ZIBDMS ähnlichen Datenmodell, sowie die relationalen Datenbank-

[23] *Peer-to-Peer*
[24] http://www.storagebox.org

systeme MySQL[25] und SQLite[26] unterstützt. Bei den relationalen Datenbanken muss jedoch das Datenmodell des ZIBDMS in ein Modell der Datenbank gewandelt werden. Der Zugriff auf Objekte eines ZIBDMS-Servers kann entweder direkt über native Protokolle oder indirekt über einen anderen ZIBDMS-Server erfolgen.

***Datenabstraktionschicht**:* Ziel dieser Schicht ist es, ein einheitliches API für die höheren Schichten zu bieten, um einen Zugriff auf Daten und Metadaten unabhängig von deren Speicherort bzw. deren Implementierung zu ermöglichen. Die Abstraktion der Metadaten wird hier, wie oben erwähnt, als Objekt bezeichnet. Abb. 16 zeigt in dieser Schicht ein Replikatmanagement. Dieses soll die Abbildung von Metadatenobjekten auf ihre tatsächlichen Speicherorte realisieren. Zurzeit wird diese Funktion nur durch den Metadatenkatalog und spezielle Objekte unterstützt. Außerdem bietet diese Schicht noch Komponenten zum einfachen und transparenten Datenzugriff (*File Access*) und zur Datenübertragung (*File Transfer*). Die Datenzugriffs-Komponente ermöglicht das Anlegen und Lesen von Dateien. Zum Kopieren von Replikaten zwischen verschiedenen Speicherorten kann die Datenübertragungskomponente benutzt werden. Es existiert ein eigenes Zugangsprotokoll „cdtp://“ zum Zugriff auf lokale Kopien von Dateien. Es gibt auch die Möglichkeit, Datentransfers zu einem festgelegten Zeitpunkt zu starten und zu beenden, um beispielsweise durch Nutzung von *Happy-Hour*-Zeiten Kosten bei der Übertragung zu sparen.

***Logische Schicht**:* In dieser Schicht werden alle Objekte des ZIBDMS modelliert. Darüber hinaus erfolgt eine Abbildung dieser Objekte auf Objekte des Metadatenkatalogs und des Replikatkatalogs bzw. auf Dateioperationen.

***Darstellungsschicht**:* Die Darstellungsschicht ermöglicht die oben beschriebenen Sichtweisen auf das System. Dazu gehören die hierarchiebasierte Verzeichnissicht sowie die Attributsicht. Zusätzlich bietet sie ihre Funktionalitäten über eine spezielle Schnittstelle – das ZIBDMS-API – höheren Schichten und anderen Anwendungen an.

***Kontrollschicht**:* Diese Schicht erlaubt den Zugriff von außen auf das System. Dazu werden verschiedene Schnittstellen wie etwa Corba[27], SOAP[28] oder NFS dem Benutzer bzw. seinen Anwendungen zur Verfügung gestellt. Im Rahmen der Arbeit von [48] wurde ein NFS-Loopback-Server in das ZIBDMS implementiert, der einen Zugriff auf die Daten durch einen unmodifizierten NFS-Client ermöglicht.

---

25 http://www.mysql.com
26 http://www.sqlite.org
27 *Common Object Request Broker Architecture*, http://www.corba.org.
28 *Simple Object Access Protocol*, http://www.w3.org/TR/soap/.

6 Ein Netzwerkinformationsdienst: Delphoi

In Kapitel 4 wurde ein einfaches Prinzip zur Beschreibung von Aufträgen, Daten und Ressourcen mittels ClassAds vorgestellt. Außerdem wurde beschrieben, wie zueinander passende ClassAds zusammengefügt werden. Kapitel 5 präsentierte ein Datenmanagementsystem. Es wurden Möglichkeiten aufgezeigt, Informationen über Daten zu erfragen und auf Daten zuzugreifen. Damit wäre es möglich, Aufträge, Daten und Ressourcen so einander zuzuordnen, dass ein Job ausgeführt werden kann.

Ziel des Resourcebrokers soll aber sein, Daten und Ressourcen möglichst optimal im Sinne des Auftrags zu wählen. Um aber beispielsweise die Gesamtzeit für einen Job zu berechnen, müssen Informationen über die verfügbaren Netzwerkverbindungen und deren Bandbreiten bekannt sein.

Für diese Aufgabe gibt es bestimmte Dienste, so genannte Netzwerkinformationsdienste bzw. Netzwerkvorhersagedienste. Ein solcher Dienst, *Delphoi*[29] [27, 28], wird in diesem Kapitel ausführlicher vorgestellt. Delphoi wurde gewählt, da dieser bereits im ZIBDMS für das Replikatmanagement[30] benutzt wird. Delphoi kann andere Anwendungen und deren Benutzer durch die Bereitstellung verschiedenartiger Informationen unterstützen:

- *Metadaten* mit Informationen darüber, welche Messdaten bzw. Parameter auf welchen Ressourcen gesammelt werden.
- *Messdaten* einzelner Ressourcen, wobei dabei sowohl vergangene, aktuelle als auch zukünftige Daten erfragt werden können. Bei Anfragen über die zukünftige Entwicklung bestimmter Messdaten ist Delphoi in der Lage, Vorhersagen auf der Basis der bisher gemessenen Werte zu treffen. Dazu wurde die Vorhersagebibliothek des NWS[31] implementiert.
- *Systemnahe Netzwerkinformationen*, etwa die verfügbare Bandbreite, die Kapazität[32] oder die Latenzzeit einer Netzwerkverbindung zwischen zwei Ressourcen in einem durch den Anfrager festgelegten Zeitraum. Delphoi kann das Maximum, das Minimum oder den Mittelwert der angefragten Größe liefern.
- *Höherwertige Netzwerkinformationen*, d.h. Anfragen, die von Delphoi analysiert und aus den systemnahen Informationen ermittelt werden. Beispielsweise kann Delphoi eine optimale Konfiguration einer TCP-Verbindung vorschlagen. Dazu muss bei der Anfrage lediglich die zu übertragende Da-

---

29 http://www.gridlab.org/WorkPackages/wp-7/

30 Das Replikatmanagement wird zurzeit im Rahmen einer Diplomarbeit von Monika Moser am ZIB in das ZIBDMS implementiert. Zum Thema „Replikatmanagement“ siehe u.a. auch [19].

31 *Network Weather Service*, http://nws.cs.ucsb.edu/

32 Die Kapazität bezeichnet im Gegensatz zur verfügbaren Bandbreite die größtmögliche theoretische Bandbreite zwischen zwei Ressourcen.

tenmenge, die benutzte Transfermethode und eine Startzeit angeben werden. Die Konfiguration umfasst u.a. Empfehlungen für die Größe des Sende- und des Empfangspuffers. Außerdem kann Delphoi die erwartete Übertragungszeit für eine bestimmte Datenmenge schätzen. Diese Funktion wird bei der Implementierung des Resourcebrokers helfen, die Übertragungszeiten von Dateien zu berechnen.

- Des Weiteren ist Delphoi in der Lage, *Warteschlangen* verschiedener Systeme (u.a. Condor) zu überwachen und Informationen über diese zu liefern, wie etwa die durchschnittliche Wartezeit für einen Job in einem bestimmten Warteschlangensystem.

6.1 Architektur von Delphoi

Delphoi setzt sich aus drei wesentlichen Komponenten zusammen. Kernstück ist der *Delphoi-Server*. Dieser nimmt Anfragen von anderen Anwendungen entgegen und ist für deren Bearbeitung verantwortlich. Anfragen können entweder über eine in den Server integrierte Socketschnittstelle oder über eine der *Webservice-Schnittstellen* gerichtet werden, wie es in Abb. 17 gezeigt wird.

Die zweite Aufgabe des Servers ist, die Clients auf den einzelnen Ressourcen, den *Pythias*, miteinander bekanntzumachen. Pythias sind für das Sammeln von Informationen und Messdaten verantwortlich.

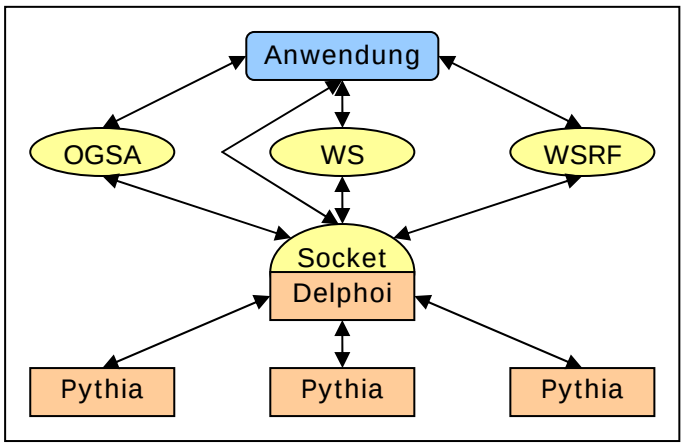


Abb. 17 Architektur von Delphoi, nach [27]
Die Pfeile zeigen Informationsflüsse zwischen den Komponenten an. Die Socketschnittstelle ist kein eigenständiges Modul, sondern im Delphoi-Server integriert.

6.1.1 Pythia

Pythias sind die Komponenten eines Delphoi-Systems, die die benötigten Informationen für Delphoi sammeln und speichern. Ein Pythia muss auf allen Ressourcen gestartet werden, für die Messdaten erhoben werden sollen. Für Res-

sourcen, die die gleichen WAN[33]-Verbindungen nutzen, reicht es aus, Pythia auf einer dieser Ressourcen auszuführen. Dadurch kann zum einen der Installationsaufwand verringert werden, zum anderen wird die Leistung der anderen Ressourcen nicht durch die Messungen und das Speichern der Daten beeinflusst. Pythias sind interne Komponenten des Systems, d.h. sie können nicht direkt durch einen Anwender sondern nur durch den Delphoi-Server abgefragt werden. Alle gesammelten und gemessenen Informationen werden lokal auf der Ressource, auf der Pythia ausgeführt wird, gespeichert. Dies entlastet das Netzwerk, da nicht benötigte Ergebnisse nicht an den Delphoi-Server geschickt werden müssen.

Zur Informationsgewinnung benutzt ein Pythia verschiedene Module, welche in der Regel externe Tools auf der jeweiligen Ressource ausführen und die Ergebnisse für Pythia konvertieren. Durch den modularen Aufbau ist es möglich, nur die benötigten Messungen auf einer Ressource ausführen zu lassen. Zudem können neue Module integriert werden. Die Festlegung, welche Module gestartet werden, kann der Besitzer der jeweiligen Ressource treffen. Einige Module, die Messungen an der Netzwerkverbindung zwischen der eigenen und einer anderen Ressource vornehmen, benötigen als Kommunikationspartner auf der anderen Ressource das gleiche Modul. Ein Pythia kann dazu beim Delphoi-Server eine Liste aller bekannten Pythias im Netz erfragen, die auch dieses Modul besitzen.

Für die Gewinnung von Informationen über den Rechner wie z.B. dessen CPU-Auslastung nutzt ein Pythia einen weiteren GridLab-Dienst namens Mecury[34]. Informationen über das Netzwerk erhält Pythia durch vier weitere Module – *Delay*, *PathRate*, *PathChirp* und *TopoMan*.

Delay sendet ein ein-Byte-großes IP-Paket zu einer anderen Ressource und so schnell wie möglich zurück und berechnet daraus die durchschnittliche Zeit für einen so genannten *Round-Trip*.

Das Modul *PathChirp* misst die verfügbare Bandbreite zwischen zwei Rechnern mit dem Programm PathChirp. Dazu sendet es eine Serie von Paketen nach speziellen Schemata (*Chirps*). Durch eine schnelle Erhöhung der Senderate jedes Chirps kann das Programm dynamisch die verfügbare Bandbreite vorhersagen. Es basiert auf dem Prinzip der *selbstverursachten Überlastung*[35]. Dieses macht die folgende Annahme: Wenn der Netzwerkverkehr zu hoch ist, werden die Pakete an einem Router vorübergehend in dessen Warteschlange gehalten werden, wodurch sich die Übertragungszeit, verglichen mit Paketen, die mit einer geringen Senderate gesendet wurden, erhöht [34].

PathRate ermittelt die Kapazität, also die maximal mögliche Bandbreite zwischen zwei Ressourcen unter Nutzung des Programms PathRate. Dazu nutzt es

---

33 Wide Area Network

34 http://www.gridlab.org/WorkPackages/wp-11/

35 *self-induced congestion*

Paketverteilungsmethoden [15] und statistische Techniken, um den Flaschenhals einer Netzwerkverbindung zu identifizieren.

TopoMon wird benutzt, um Informationen über die Netzwerktopologie zwischen Rechnern zu erhalten, ähnlich wie es Traceroute liefert. Im Gegensatz zu den anderen Netzwerk-Modulen von Pythia benötigt TopoMon auf der Gegenseite kein äquivalentes Modul.

Darüber hinaus bietet Pythia höhere Module an, welche kompliziertere Anfragen beantworten, indem sie diese zerlegen und auf die oben beschriebenen Module zurückgreifen.

Um Fehler bzw. fehlerhafte Messdaten zu vermeiden, kann auf einem Rechner immer nur eine Messung gleichzeitig stattfinden. Denkbar wäre eine gleichzeitige Bandbreitenmessung zweier Module auf der gleichen Ressource, was ein reduziertes Ergebnis zur Folge haben könnte. Kommt es trotzdem zu einem Fehler, weil z.B. der andere Rechner gerade eine Messung durchführt und das gewünschte Modul nicht gestartet wurde[36], gibt es zwei Möglichkeiten. Bei Modulen, die häufig (etwa jede Viertelstunde oder häufiger) gestartet werden, wird bis zur nächsten Ausführung gewartet. Ist die Frequenz jedoch gering, wird der Rechner vermerkt und der Test später ausgeführt. Die Frequenz teilt jedes Modul dem Pythia-Client mit.

6.1.2 Delphoi-Server

Der Delphoi-Server dient den einzelnen Pythias als Kontaktpunkt, über ihn können sie eine Liste aller verfügbaren Pythias im Netz erfragen. Außerdem ist der Server der einzige Kommunikationspartner für Anfragen anderer Anwendungen. Diese Anfragen können entweder über die in den Server integrierte Socketschnittstelle oder über eine der Webserviceschnittstellen gerichtet werden. Letztere sind eigenständige Komponenten, die auf der Socketschnittstelle aufbauen. Zurzeit werden ein proprietärer Webservice sowie die Standards OGSA[37] und WSRF[38] unterstützt. Dank der Modularität können weitere Schnittstellen implementiert werden.

Das Delphoi-Modul selbst enthält sämtliche Logik, um Anfragen in einfache Messwerte umzuwandeln und die jeweils benötigten Pythias zu kontaktieren. Im folgenden Abschnitt findet sich ein Beispiel, das diese Konvertierung beschreibt.

---

[36] Dies ist möglich, da das Scheduling der einzelnen Module nur lokal erfolgt.

[37] *Open Grid Service Architecture*, Spezifikation für Grid Services auf der Basis von Webservices. http://www.globus.org/ogsa/.

[38] *Web Services Request Framework*, Weiterentwicklung von OGSA. http://www.globus.org/wsrf/.

Der Server ist, wie alle anderen Komponenten, in Java implementiert, was das System betriebssystem- und rechnerarchitekturunabhängig macht. Zu beachten ist jedoch, dass die Pythias auf externe Programme zurückgreifen, die auch für die jeweiligen Ressourcen vorhanden sein müssen.

6.2 Beispielanfrage

Die Funktionsweise von Delphoi und Pythia wird hier an einem Beispiel zusammengefasst. Es wird die Anfrage gestellt, wie lange es dauert, eine Datei von einem Rechner zu einem anderen zu übertragen.

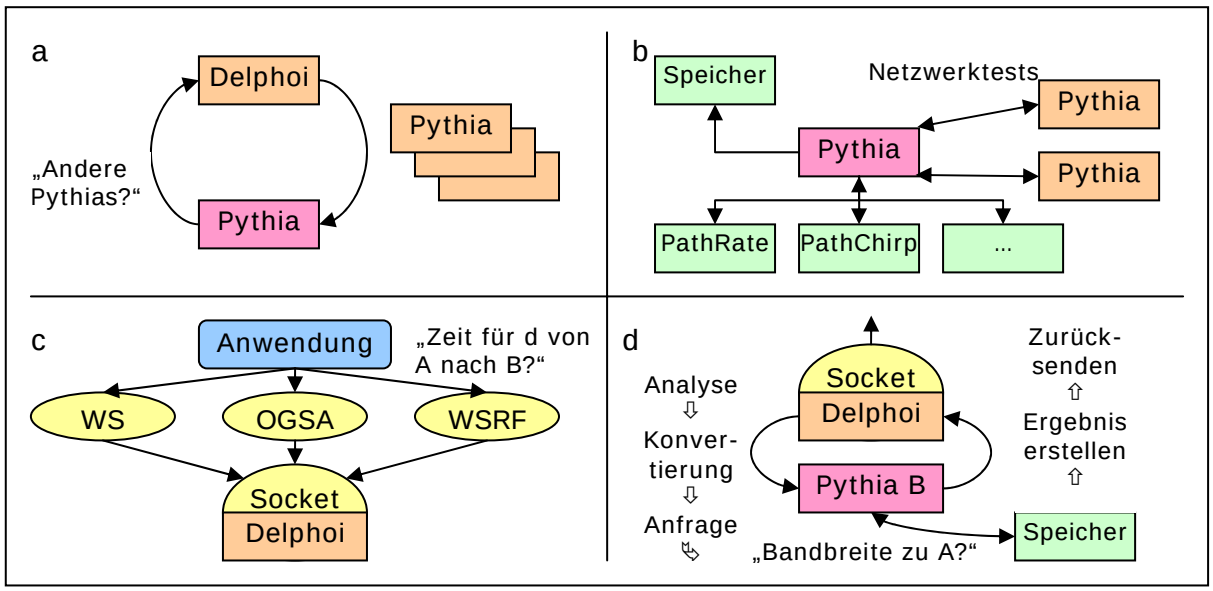


Abb. 18 Funktionsweise von Delphoi, nach [27]

Bevor der Anwender eine Anfrage stellen kann, müssen die notwendigen Informationen gesammelt werden. Dazu starten die Pythias periodisch einzelne Module. Netzwerktests benötigen mindestens einen weiteren Pythia, daher kann ein Pythia eine Liste aller Pythias beim Delphoi-Server erfragen. (Abb. 18a).

Pythia A: Hole Liste aller Pythias von Delphoi.

Anschließend kann Pythia die einzelnen externen Tools über die Module starten und Messungen auf dem Rechner und im Netzwerk ausführen (Abb. 18b).

Pythia A: Messe Bandbreite zu Rechner B.

Anwender können Anfragen nur direkt an Delphoi über eine der verfügbaren Schnittstellen (Sockets, proprietäre Webservices, OGSA, WSRF) richten (Abb. 18c).

Anfrager: Sende ‚Wie lange benötigt der Transfer einer 1 MB großen Datei von Rechner A zu Rechner B?‘ an Delphoi.

Delphoi analysiert die Anfrage und kontaktiert den zuständigen Pythia. Dieser liest die benötigten Informationen aus dem lokalen Speicher aus und sendet sie an Delphoi zurück, welcher die Informationen verarbeitet und dem Anfrager die Ergebnisse zurücksendet (Abb. 18d).

Delphoi: Übertragungszeit ist von Bandbreite zwischen A und B abhängig. Frage bei Pythia A die Bandbreite zu B an.

Pythia A: Hole Bandbreiteninformation zu Rechner B aus lokalem Speicher und sende Information an Delphoi zurück.

Delphoi: Berechne aus Dateigröße und Bandbreite die benötigte Übertragungszeit. Sende Ergebnis an Anfrager zurück.

7 Verwandte Projekte auf ClassAd-Basis

Verschiedene existierende Brokersysteme wurden bereits in der vorangegangenen Studienarbeit miteinander verglichen. Daher werden in diesem Kapitel Projekte präsentiert und bewertet, die auf dem ClassAd-Mechanismus aufbauen. Es sollen u.a. Vorteile der neuen ClassAd-Version, die zwei der drei vorgestellten Projekte benutzen, erklärt werden aber auch, warum sich die hier vorgestellten Projekte nicht für die Realisierung des Modells eignen.

7.1 Gangmatching

Gangmatching ist eine Erweiterung des bilateralen Matchmakings von Condor durch das Condor-Team [30, 32]. Grundannahme ist, dass man häufig mehr als zwei Komponenten zur Ausführung eines Jobs benötigt. Beispielsweise benötigt man zur Nutzung einer bestimmten Software eine Lizenz, wobei deren Anzahl ist und nicht jede Lizenz für jeden Rechner zur Verfügung steht. Man sucht also neben einem ausführenden Rechner, der die Anforderungen des Benutzers erfüllt, auch eine für diesen passende Lizenz.

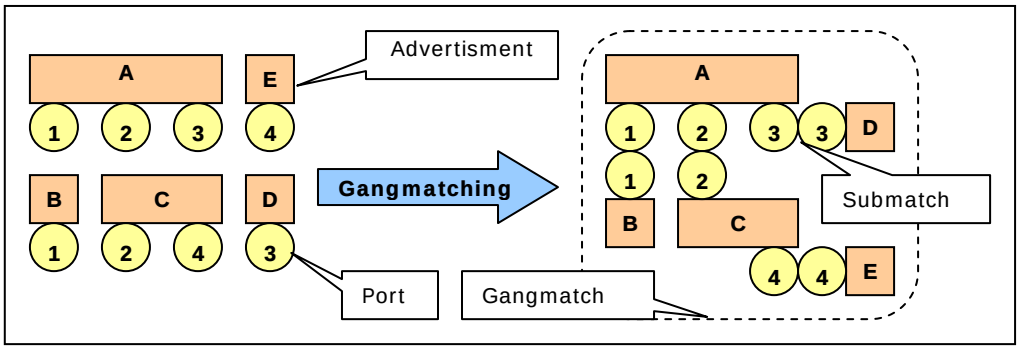


Abb. 19 Ablauf des Gangmatching nach [30]
Die verschiedenen Zahlen der Ports sollen verschiedene Anforderungen darstellen. Der Match ist vollständig, wenn alle Ports gematcht werden konnten.

Dieses Problem kann auch mit dem bilateralen Matchmaking gelöst werden. Dann müsste man zunächst einen passenden Rechner suchen und diesen reservieren, um anschließend mittels einer neuen Anfrage eine Lizenz zu suchen. Ist keine Lizenz für den gewählten Rechner vorhanden, würde der Rechner unnötig blockiert werden. Das gleiche gilt, wenn erst eine Lizenz und dann eine Rechner ausgewählt wird.

Beim Gangmatching[39] wird das von Condor bekannte implizite bilaterale Matchmaking ersetzt durch ein eine Liste mit expliziten bilateralen Matches. Implizit bedeutet, dass Condor nur zwei Typen von ClassAds kennt, *Jobs* und *Ma-*

[39] Gangmatching funktioniert mit beliebigen Typen von ClassAds und ist nicht auf Ressourcen und Lizenzen begrenzt.

chines und daher immer automatisch eine Job- und eine Machine-ClassAd matcht, ohne dass dies explizit in einer der ClassAds angegeben werden muss. Gangmatching hingegen kennt keine festen Typen von ClassAds. Daher muss im **Constraint**-Attribut[40] immer der Typ der zu matchenden ClassAd angegeben werden.

Jede ClassAd enthält ein **Ports**-Attribut (siehe auch Abb. 19), welches eine Liste aus einer oder mehreren ClassAds umfasst. Listen sowie das Schachteln von ClassAds sind Merkmale der neuen ClassAd-Version. Beim Matchmaking wird nicht die gesamte ClassAd gematcht, sondern nur deren Ports. Wurde ein Port noch nicht gematcht, so bezeichnet man diesen als „offen“. Der Match ist abgeschlossen, wenn alle Ports aller beteiligten ClassAds gematcht wurden. Der Algorithmus ordnet die Job-ClassAds nach ihrer Priorität und versucht mittels eines Top-down-Backtracking-Algorithmus die Ports aller beteiligten ClassAds zu erfüllen.

```
[ Type = "Job";
  Owner = "raman";
  Cmd = "run_sim";
  Ports =
  {
  [ // request a workstation
    Label = "cpu";
    ImageSize = 28M;
    Rank = cpu.Memory/32;
    Constraint =
      cpu.Type=="Machine" &&
      cpu.Arch=="INTEL" &&
      cpu.OpSys=="LINUX" &&
      cpu.Memory>=Imagesize;
  ],

  [ // request a license
    Label = "license";
    Host = cpu.Name; // cpu name
    Rank = 0;
    Constraint =
      license.Type=="License" &&
      license.App==Cmd;
  ]
  }
]
```

```
[ Type = "Machine";
  KeybrdIdle= '00:23:12'; // h:m:s
  Disk = 323.4M; // mbytes
  Memory = 256M; // mbytes
  LoadAvg = 0.042969;
  Arch = "INTEL";
  OpSys "LINUX";
  Name = "foo.cs.wisc.edu";
  Ports = {
  [  Label = "requester";
     Rank = 1G-requester.ImageSize;
     Constraint =
       requester.Type=="Job" &&
       requester.Owner!="rival" &&
       LoadAvg < 0.3 &&
] ] }
```

```
[ Type = "License";
  App = "sim_app";
  ValidHost= "foo.cs.wisc.edu";
  Ports = {
  [ Label = "requester";
    Rank = 0;
    Constraint=
      requester.Type=="Job" &&
      requester.Host==ValidHost
] ] }
```

Abb. 20 Beispiel für eine Job- (links), eine Machine- (rechts oben) und eine License-ClassAd (rechts unten), nach [32]

Jeder Port enthält ein **Label**-Attribut, welches es ermöglicht, aus einem Port heraus auf die Attribute eines darüber stehenden Ports zuzugreifen. Labels sind von ihrer Definition bis zum Ende der Portlist gültig, daher kann ein weiter oben stehender Port nicht auf die darunter liegenden zugreifen. Dies sieht man in Abb. 20 in der Job-ClassAd auf der linken Seite im Port **license** beim Attribut **Host**,

[40] Entspricht dem **Requirements**-Attribut im Condor-System.

welches auf **cpu.name** verweist. So kann die License-ClassAd rechts unten indirekt auf **cpu.name** zugreifen.

Bewertung: Gangmatching zeigt die Möglichkeiten der neuen ClassAds mit Listen und Verschachtelungen, lässt aber globale Attribute vermissen, um über alle Ports hinweg bestimmte Eigenschaften wie z.B. die Gesamtausführungszeit optimieren zu können. Darüber hinaus stand der Quellcode des Gangmatching-Projekts bis zur Fertigstellung der Arbeit nicht zur Verfügung.

7.2 Stork

Bei Stork [22, 23] handelt es sich um einen vom Condor-Team entwickelten Zeitplaner zur Durchführung von Datenoperationen wie z.B. der Reservierung von Speicherplatz oder dem Transfer von Dateien. Die Autoren von [23] behaupten, dass sich normale Jobs und Datenaufgaben grundlegend unterscheiden und daher nicht vom einem einzigen Zeitplaner oder Resourcebroker behandelt werden sollten. Als Beispiel wird angeführt, dass der Ausfall eines Netzwerkprotokolls nicht einen Abbruch der Übertragung nach sich ziehen muss. Vielmehr könnte ein entsprechendes System die Übertragung über ein anderes verfügbares Protokoll fortführen. Eine entsprechende Funktionalität haben die meisten Broker nicht.

Darüber hinaus vereinfacht Stork Dateioperationen, indem es eine definierte Sprache auf Basis von ClassAds zur Beschreibung von Aufträgen zur Verfügung stellt. Abb. 21 zeigt drei Beispiele für solche Aufträge.

```
[
  dap_type = "reserve";
  dest_host = "db18.cs.wisc.edu";
  reserv_size = "100 MB";
  duration = "2 hours";
  reserve_id = 3;
]

[
  dap_type = "transfer";
  src_url =
"srb://ghidorac.sdsc.edu/home/kosart.condor/1.dat";
  dest_url = "nest://db18.cs.wisc.edu/1.dat";
]

[
  dap_type = "release";
  dest_host = "db18.cs.wisc.edu";
  reserve_id = 3;
]
```

Abb. 21 Drei Anfragen an Stork, aus [23]
Die oberste Anfrage reserviert Speicherplatz, die mittlere veranlasst die Übertragung einer Datei und die unterste gibt den reservierten Speicherplatz wieder frei.

Da in der Regel Berechnungsaufgaben mit Datenübertragungen verbunden sind, kann Stork mit übergeordneten Planern zusammenarbeiten, beispielsweise mit dem *Direct Acyclic Graph Manager* (DAGMan)[41]. Der Nutzer erstellt eine DAG-Spezifikation, welche alle notwendigen Aufgaben sowie die Abhängigkeiten zwischen diesen enthält. DAGMan erstellt daraus einen Baum mit Teilaufgaben und leitet diese entsprechend ihres Typs weiter. Dazu wurde der Manager so erweitert, dass er zwischen Datenoperationen und normalen Jobs unterscheiden kann – Berechnungsaufgaben werden an Condor bzw. Condor-G übergeben, Datenübertragungsaufgaben an Stork.

Bewertung: Nachteilig ist, dass Stork nur mit Datenübertragungsaufgaben umgehen kann. Für weitergehende Aufgaben benötigt man DAGMan. DAGMan führt aber keine globalen Optimierungen durch, d.h. er achtet nicht auf Beschränkungen wie eine möglichst kurze Gesamtausführungszeit. Eine Anpassung von Stork ist nicht möglich, da die gesamte Architektur auf dem Ansatz beruht, dass Daten- und Jobaufträge getrennt voneinander zu behandeln sind. Das Ersetzen von DAGMan durch einen anderen Planer löst das Problem ebenfalls nicht, da die Wahl der einzelnen Ressourcen und Datenspeichern erst während der einzelnen Teilaufgaben durchgeführt wird, der Manager aber die Aufgabenreihenfolge davor festlegen muss.

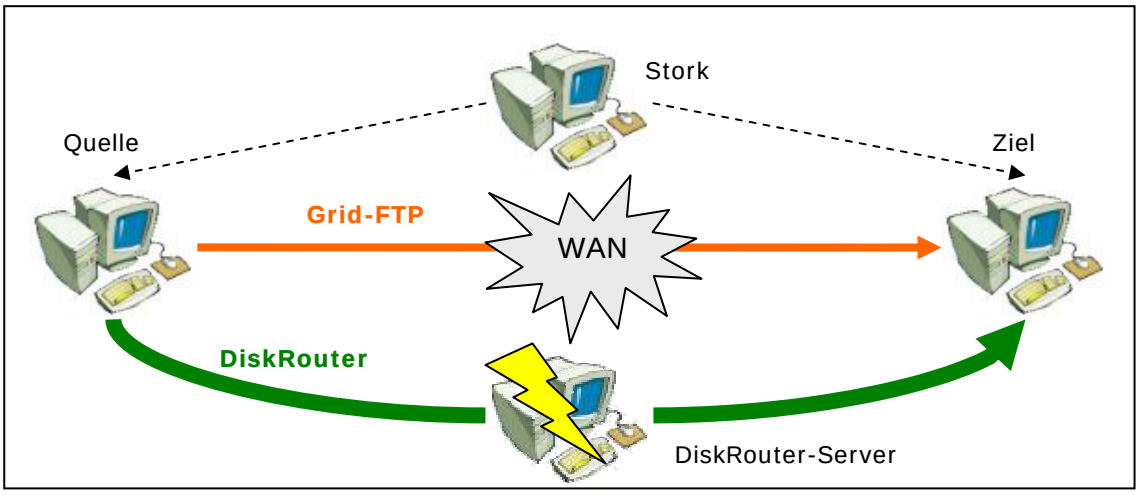


Abb. 22 Beispiel für den dynamischen Protokollwechsel von Stork, nach [22]
Gestrichelte Pfeile geben Steuerungsflüsse an, die anderen Pfeile zeigen Datenübertragungen über verschiedene Protokolle an. Je dicker einer dieser Pfeile, desto schneller ist dieses Protokoll.

Vorteile bietet Stork durch die große Zahl von unterstützten Datenübertragungsprotokollen, u.a. FTP, GridFTP, HTTP, DiskRouter[42] sowie diversen Datenspeichersystemen. Stork kann auch dynamisch während einer Übertragung das

41 http://www.cs.wisc.edu/condor/dagman
42 http://www.cs.wisc.edu/condor/diskrouter/

benutzte Protokoll wechseln, z.B. zunächst ein schnelles, aber nicht unbedingt zuverlässiges Protokoll verwenden und im Falle einer Unterbrechung der Verbindung eine langsamere aber zuverlässigere Alternative benutzen [22]. Abb. 22 zeigt eine solche Situation. Zunächst wird das schnellstmögliche Protokoll DiskRouter verwendet. Alle Datentransfers mit diesem Protokoll gehen über einen speziellen DiskRouter-Server. Fällt dieser Rechner aus, so kann Stork auf das (in diesem Fall) langsamere GridFTP wechseln, welches eine Direktverbindung zum Zielrechner aufbaut.

Des Weiteren ist Stork über ein Optimierungssystem in der Lage, die Parameter von Netzwerkverbindungen zu optimieren, indem beispielsweise die Paktegröße oder die Puffergröße angepasst wird.

7.3 SAMGrid

Das SAMGrid [6] ist aus einer Zusammenarbeit des Fermi National Accelerator Laboratory und dem Condor-Team entstanden, bei der zwei verschiedene Softwaresystem – *SAM* und *Condor-G* – kombiniert wurden.

SAM[43] ist ein verteiltes System ur Verwaltung von Daten in einem Grid. Es ist jedoch nicht möglich, mit SAM Gridjobs zu erstellen bzw. auszuführen.

Condor-G [41] ist eine Erweiterung von Condor für das Grid Computing. Sie ermöglicht eine fehlertolerante Jobübermittlung und kann auf Ressourcen über das *Globus Resource Allocation Manager*-Protkoll (GRAM) zugreifen. GRAM ist eine Komponente des Globus Toolkits[44], die die Ausführung von Jobs auf Ressourcen ermöglicht. Es hat den Vorteil, dass es sicher ist und eine einheitliche Schnittstelle zu vielen verschiedenen Batchsystemen bietet. Es besitzt aber kein benutzerfreundliches Interface und kein zuverlässiges Protokoll zur Übertragung von Jobs und Daten [6]. Condor-G erweitert GRAM um diese fehlenden Funktionen. Eine ausführlichere Betrachtung von Condor-G findet sich in der vorangegangen Studienarbeit.

SAMGrid stellt eine Verbindung zwischen Condor-G und SAM dar. Ziel soll es sein, dass Condor-G Jobs so plant, dass möglichst wenige Daten vor einem Job übertragen werden müssen, um so die Ausführungszeit zu verkürzen. Es stellt sich die Frage, ob dieses Problem nicht mit Condor bzw. Condor-G gelöst werden kann. Es sind weiterhin nur zwei ClassAds vorhanden, die gematcht werden müssen und die benötigten bzw. vorhandenen Dateien könnten einfach in die jeweiligen ClassAds mit aufgenommen werden. Bedenkt man aber, dass Datenmangementsysteme oftmals hunderttausende Dateien verwalten, so würde das Veröffentlichen aller Dateien in einer ClassAd das System unskalierbar machen [6].

---

43 *Sequential data Access via Meta-data*, http://d0db.fnal.gov/sam/

44 http://www.globus.org

Weiterhin problematisch ist, dass Condor-G nicht einfach auf Informationen zurückgreifen kann, die nicht zentral in einem Directory Service vorliegen. Ein Beispiel wäre die Anfrage: „Ich habe eine Liste mit den Dateien d1, d2, …, d500. Wähle die Ressource aus, die die meisten dieser Dateien enthält.“ So eine Anfrage kann nur sehr umständlich mit ClassAds beschrieben werden. Daher wurden Condor-G und Condor so erweitert, dass benutzerdefinierte Funktionen in ClassAds benutzen werden können, welche zum Zeitpunkt des Matchens aufgerufen werden. Abb. 23 zeigt zwei typische ClassAds von SAMGrid. Im **Rank**-Attribut der JobClassAd auf der rechten Seite sieht man so eine Funktion **sam_rank_overlap**, die den Prozentsatz zurückgibt, inwieweit die benötigten Dateien und die auf dem Rechner vorhanden Dateien übereinstimmen.

```
[
  MyType = "Machine";
  TargetType = "Job";
  Software_CAF = "Installed";
  site = "UToronto";
  schema = "v0_8";
  SamStationName = "cdf-toronto";
  SamStationUniverse = "prd";
  SamStationExp = "cdf";
  Name = "cdf-toronto.cdf.prd";
  architecture = "Linux+2.4";
  gatekeeper_url =
    "XXXXX:2119/jobmanager";
  nameservice = "IOR:000…44c3a…65000";
]
```

```
[
  MyType = "Job";
  TargetType = "Machine";
  Rank =
    sam_rank_overlap(
      other.SamStationName,
      "jbot0h-361404",
      other.nameservice);
  Requirements =
    other.Software_CAF isn't Undefined
    && other.SamStationUniverse=="prd"
    && other.SamStationExp == "cdf";
  GlobusResource =
    "$$(gatekeeper_url)";
  Cmd = …
  Args = …
]
```

Abb. 23 Beispiel für Job- und MachineClassAd in SAMGrid, nach [6]
Zu beachten ist das Rank-Attribut der rechten ClassAd, welches eine benutzerdefinierte Funktion aufruft.

Somit ist man beim Matchmaking nicht mehr nur auf die statischen, im Vornherein festgelegten Angaben innerhalb der ClassAds angewiesen. Es ist jedoch zu beachten, dass sich die Leistung des Matchmakers verringern kann, wenn die benutzerdefinierten Funktionen externe Dienste abfragen, etwa durch die Latenzzeiten der benutzen Netzwerkverbindungen. Zur Lösung des Problems bieten sich Cachingstrategien an.

Der Ablauf der Jobplanung entspricht zunächst dem Matchmaking von Condor. Der Anwender übermittelt seine Jobbeschreibung an einen Condor-G-Agenten, der diese in eine ClassAd umwandelt und an das Warteschlangensystem von Condor-G weiterreicht. Die Beschreibung enthält das auszuführende Programm, die Namen der benötigten Dateien und die vollständigen Pfade zu den einzelnen Dateien (siehe Abb. 24, Schritt 1). Das Warteschlangensystem schickt die ClassAd an den Matchmaker weiter, sobald dieser verfügbar ist (2). Die verfügbaren

Grid Sites, also Verbünde aus Ressourcen und Datenspeichern, die nahe[45] beieinander sind, veröffentlichen ihre Eigenschaften in Form von ClassAds, die sie über die Grid-Sensoren an den Matchmaker weiterleiten. Der Matchmaker prüft anschließend, ob JobClassAd und Grid Site-Beschreibung gematcht werden können. Dabei fragt er für jede Grid Site über die benutzerdefinierten Funktionen beim jeweiligen Datenmanagementsystem an, über welche Daten die Site verfügt (3).

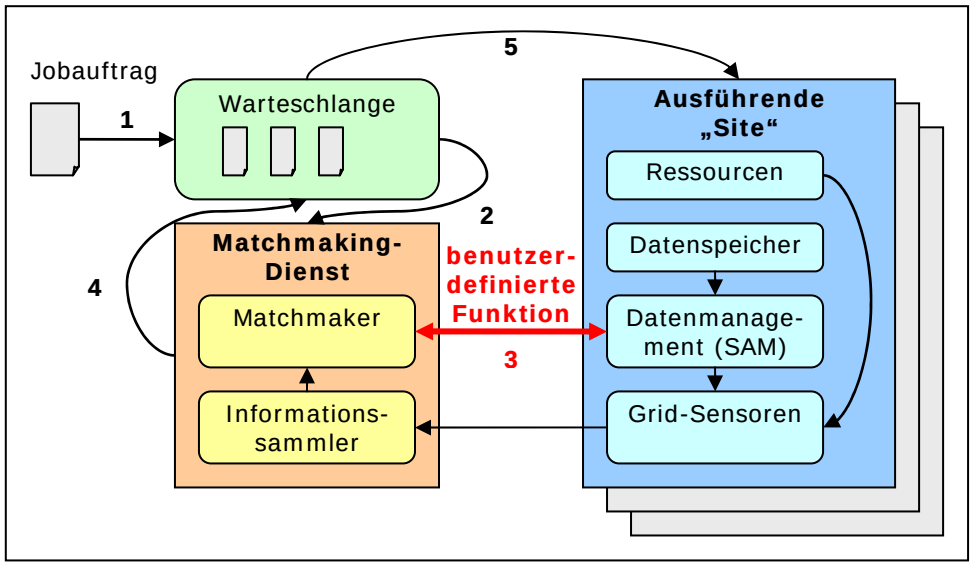


Abb. 24 Architektur des Jobmanagement in SAMGrid, nach [6]
(1) Der Job wird an das Warteschlangen-System übermittelt. (2) Der Matchmaker versucht, die Jobbeschreibung mit den Informationen der Grid-Sensoren zu matchen. (3) Er kontaktiert das Datenmanagementsystem über eine benutzerdefinierte Funktion. (4) Der Matchmaker übergibt den Match an das Warteschlangen-System. (5) Der Job wird an die gewählte Ressource übertragen und ausgeführt.

Auf Basis der Ergebnisse kann der Matchmaker die beste[46] Grid Site auswählen und Condor-G über die Wahl informieren. Daraufhin kann Condor-G die gewählte Grid Site und weitere Informationen der JobClassAd hinzufügen (4). Anschließend kann Condor-G die Ausführung des Job veranlassen, indem der Job per GRAM-Protokoll an die nun Condor-G bekannte Grid Site geschickt wird (5).

Bewertung: Das SAMGrid ist ein Resourcebroker, der die Verteilung der Daten mit in die Jobauswahl einbezieht. SAMGrid führt den Job immer auf der Ressource aus, die die meisten verfügbaren Daten schon vor der Jobausführung besitzt. Im Modell des in dieser Arbeit entwickelten Brokers gilt jedoch die Annahme, dass Datenspeicher und ausführende Ressourcen (Rechner) zu unterscheiden sind, da sich ihre Anforderungen stark voneinander unterscheiden. SAMGrid matcht daher nicht Rechner sondern *Grid Sites*. Durch dieses Verfahren verein-

[45] „Nahe“ bedeutet hier im gleichen LAN befindlich und über die gleiche WAN-Verbindung angebunden.
[46] D.h. diejenige mit dem höchsten Rangwert.

facht sich die datenabhängige Jobplatzierung, da es genügt, alle Datenspeicher anhand der Anzahl der vorhandenen Dateien zu sortieren.

Der Ansatz dieser Arbeit geht darüber hinaus, da eine ausführende Ressource keine der benötigten Dateien besitzen muss. Vielmehr erfolgt die Wahl von Datenspeichern und Ressourcen aufgrund bestimmter Anforderungen des Auftraggebers. Sollte eine passende Ressource viele der benötigten Dateien besitzen, so wird diese mit hoher Wahrscheinlichkeit auch zur Ausführung kommen, da die Transferzeiten für diese Dateien gleich Null sind.

In [6] findet sich keine Aussage dazu, wie die benötigten Dateien, die nicht auf der gewählten Ressource vorhanden sind, zu dieser gelangen. So müssten zunächst Datenspeicher gesucht werden, die diese Dateien besitzen. Anschließend müssten diese Dateien zur ausführenden Ressource übertragen werden. Werden die Datenspeicher jedoch nicht von Beginn an mit in die Auswahl einbezogen, so kann eine optimale Gesamtausführungszeit nicht garantiert werden.

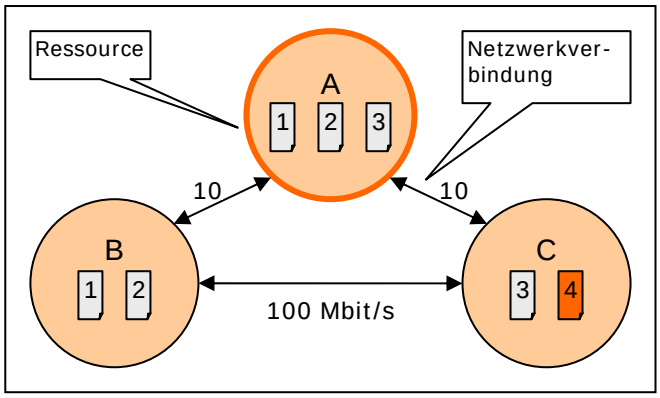


Abb. 25 Beispiel für ungünstige Ressourcenauswahl bei SAMGrid

Abb. 25 zeigt ein Beispiel für eine solche ungünstige Auswahl. Angenommen, es soll ein Job ausgeführt werden, die die gleichgroßen Dateien (1), (2), (3) und (4) benötigt. Alle drei Ressourcen A, B und C sind für den Job geeignet. SAMGrid würde nun Ressource A wählen, da sie die meisten der benötigten Dateien besitzt. Datei (4) müsste dann von Ressource C zu Ressource A über die langsamere 10 Mbit/s-Verbindung übertragen werden. Würde Ressource C gewählt werden, so müssten zwar zwei Dateien von B zu C übertragen werden, die Verbindung ist jedoch zehnmal schneller. Insgesamt würde also der Datentransfer schneller gehen als bei der Auswahl von SAMGrid.

Eine Erweiterung von SAMGrid so, dass Datenspeicher und ausführende Ressource nicht identisch sein müssen, setzt voraus, dass der Matchmaker Ressourcen und Datenspeicher unterscheiden kann. Dann müsste dieser aber mindestens drei ClassAds gleichzeitig matchen können, was mit Condor bzw. Condor-G aufgrund der alten ClassAd-Sprachversion nicht möglich ist.

Eine sehr interessante Erweiterung sind die benutzerdefinierten Funktionen, die es ermöglichen, während des Matchmakings zusätzliche Informationen einzuholen. Die aktuelle Implementation der benutzerdefinierten Funktionen in Condor im Rahmen von SAMGrid ist leider nicht dokumentiert. Zudem gestattet sie nicht das Manipulieren von Attributen in ClassAds. Es können vielmehr nur einfache Typen zurückgegeben werden. Auch gibt es für die alte ClassAd-Version kein API, so dass die Auswertung von als Parametern übergebenen ClassAds sehr aufwendig ist.

8 Architektur des Resourcebrokers

Im Folgenden wird der Aufbau des Resourcebrokers beschrieben. Wie zu Beginn der Arbeit erklärt, soll der Broker auf dem Matchmaking-Prinzip von Condor aufbauen. Ziel des Brokers soll es sein, Jobs, Ressourcen und Daten so zu wählen, dass die Wahl bezüglich der Gesamtkosten bzw. der Gesamtausführungszeit optimal ist. Der Resourcebroker setzt sich aus vier Komponenten zusammen, dem Matchmaker als zentrale Komponente sowie drei Clients als Schnittstelle zu Jobs, Ressourcen und Daten (siehe Abb. 26). Die Daten werden dabei durch ein Datenmanagementsystem verwaltet, welches mit Hilfe des Replikatmanagements und eines externen Netzwerkinformationsdienstes das beste Replikat einer für den Job benötigten Datei heraussuchen soll.

Die Architektur des Resourcebrokers wird im Abschnitt 8.3 ausführlich beschrieben. Zuvor finden sich im folgenden Abschnitt Gründe, einen neuen Resourcebroker zu entwerfen. Der Abschnitt 8.2 beschreibt die nötigen Erweiterungen eines Datenmanagementsystems am Beispiel des ZIBDMS.

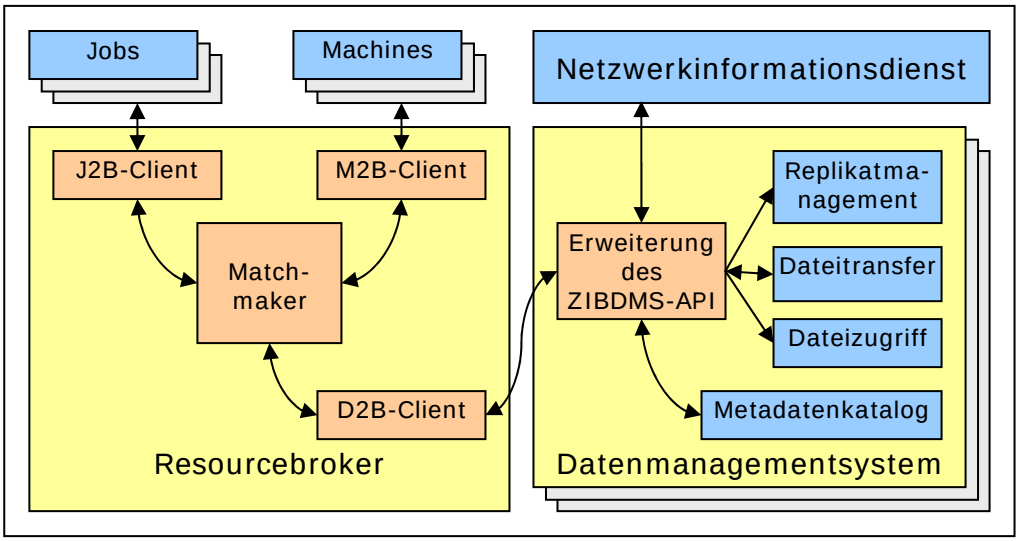


Abb. 26 Architektur- und Kommunikationsüberblick über den Resourcebroker
Blaue Bereiche stellen schon vorhandene Komponenten dar, orangefarbene kennzeichnen Komponenten, die im Rahmen der Arbeit entwickelt werden. Pfeile stellen Informationsflüsse dar, die in den folgenden Abschnitten erklärt werden.

8.1 Gründe für einen neuen Resourcebroker

Zunächst war geplant, das Condor System so zu erweitern, dass es die Verteilung von Daten und deren Übertragungszeiten mit in das Matchmaking einbezieht. Die Funktionalität von Condor hat sich für diese Funktionalität als zu beschränkt gezeigt, die Gründe dafür werden im Folgenden aufgezeigt.

Condor unterstützt zurzeit und auch in der näheren Zukunft nur die alte ClassAd-Version. Dadurch ist nur das bilaterale Matchmaking von ClassAds möglich.

Hauptziel soll es aber sein, zusätzlich zu einer Ressource auch passende Datenquellen zu finden. Diese müssten dann aber zusammen mit allen verfügbaren Dateien mit in der MachineClassAd aufgelistet sein, was das System unskalierbar machen würde. Eine andere Möglichkeit wären zwei Matches hintereinander. Zunächst sucht man eine passende Ressource für den Job und dann passende Datenspeicher für diese Job-Ressourcen-Kombination. In diesem Fall müsste aber die gewählte Ressource solange reserviert werden, bis der Match mit dem Datenspeicher abgeschlossen ist. Dadurch werden Ressourcen unnötig blockiert und die Effizienz des Systems sinkt. Zum anderen bildet die Suche nach einer Ressource und die Suche nach Datenspeichern für einen Job eine logische Einheit ähnlich einer Transaktion in einem Datenbanksystem und sollte nicht in Teile zerlegt werden.

Darüber hinaus fehlen viele nützliche Funktionen wie etwa Listen oder verschachtelte ClassAds, die ein multilaterales Matchmaking überhaupt erst ermögliche, wie es z.B. beim Gangmatching (siehe Abschnitt 7.1) benutzt wird.

Die Implementation eigener Funktionen gestaltet sich umständlich. Wie oben erwähnt gibt es eine undokumentierte Möglichkeit, eigene Funktionen in Condor zu nutzen. Es kann jedoch nicht garantiert werden, dass diese Funktion auch in zukünftigen Versionen noch vorhanden ist. Die benutzerdefinierten Funktionen können zudem nur einfache Datentypen wie zurückgeben, da die alte ClassAd-Version ClassAds als Attribute nicht zulässt. Meist kann das Ergebnis einer Anfrage aber nicht in einen solchen einfachen Datentyp kodiert werden. Ein Beispiel wäre die Anfrage an ein DMS, welches Replikat einer Datei am schnellsten zu einer bestimmten Ressource transferiert werden kann. Der Name des Datenspeichers, der am ehesten geeignet ist, reicht für die weitere Verarbeitung nicht aus. So muss der Resourcebroker unter Umständen die Gesamtausführungszeit des Jobs berechnen und braucht daher auch die erwartete Übertragungszeit. Somit muss die Anfrage zwei Werte zurückliefern. Eine Alternative, bei der die erste Anfrage nur eine ID zurückgeliefert, die bei anschließenden Anfragen mit angegeben werden muss, wurde verworfen, da dies die Zahl der benutzerdefinierten Funktion stark erhöhen würde (im Beispiel von einer auf drei Funktionen). Des Weiteren müsste auch sichergestellt sein, dass die Funktion, die die ID liefert, immer vor den anderen, von der ID abhängigen Funktionen gerufen wird.

Ein weiterer Grund, der gegen die Verwendung von Condor spricht, ist der Mangel eines API für ClassAds, wie es die neue Version bietet. D.h. selbst wenn es möglich sein sollte, ClassAds als Parameter einer benutzerdefinierten Funktion zu übergeben, so müssten diese zunächst in einem eigenen ClassAd-Modell nachgebildet werden.

Wenn Condor nicht als erweitere Basis geeignet ist, so könnte versucht werden, einen *Datenbroker* zu implementieren, der parallel zu Condor läuft. Hier gibt es zwei Möglichkeiten: Entweder wird der Job zunächst an den Datenbroker übermittelt, um dort die beste Datenquelle zu suchen. Anschließend wird diese Datenquelle in die ClassAd eingetragen und an das Condor-System weitergeleitet. Dies ist nicht realisierbar, da der Datenbroker zur Wahl des besten Replikats das Ziel eines möglichen Datentransfers kennen muss. Bei der zweiten Möglichkeit würde der Job zunächst an Condor übertragen werden, wo versucht werden würde, diesen mit einer Ressource zu matchen. Während des Matchmakings müsste dann über benutzerdefinierte Funktionen für jede benötigte Datei der Datenbroker kontaktiert werden. Der Datenbroker müsste dann die beste Quelle für die Datei suchen und diese zusammen mit der erwarteten Übertragungszeit zurücksenden. Es treten dann aber die oben erwähnten Probleme auf, da eine Funktion nur einfache Datentypen zurückgeben kann. Ein weiteres Problem ist, dass im Falle eines Matches die Datentransfers gestartet werden müssen. Condor kennt dazu nur das Attribut **`transfer_input_files`**, welche alle Dateien, die als Wert angegeben werden, vor einem Job durch eigene Protokolle überträgt. Ein Ansprechen von Datenmanagementsystemen ist dabei nicht möglich, auch kann kein Zeitpunkt gewählt werden, zu dem eine Datei übertragen werden soll.

Daher wird für den Resourcebroker dieser Arbeit auf das Condor-System verzichtet. Als Basis dient vielmehr das API der neuen ClassAd-Sprachversion. Dadurch wird es erforderlich, alle grundlegenden Komponenten von Condor selbst zu implementieren. Es wird aber versucht, wesentliche Prinzipien und Mechanismen von Condor beizubehalten, etwa die fest definierten Attributnamen. Ziel ist, dass die Erweiterungen in Zukunft, wenn Condor die neue ClassAd-Version unterstützt, einfach in das System übernommen werden können und damit auch produktiv einsetzbar sind, ohne dass die grundlegende Architektur von Condor geändert werden müsste.

Wie weiter oben erklärt, liegt die Besonderheit des Resourcebrokers darin, neben den Eigenschaften der ausführenden Ressource auch die der benötigten Daten mit zu berücksichtigen. Es soll daher die Möglichkeit geben, verschiedene Bedingungen zu wählen und zu kombinieren, die sich auf den gesamten Job inkl. aller notwendigen Datentransfers und nicht nur auf die eigentliche Ausführung beziehen. Dies erfordert globale Kenntnisse über die einzelnen Teile des Jobs.

Auch wenn die in Kapitel 7 vorgestellten Projekte schon Ansätze zeigen, die über den bilateralen Match hinausgehen, so bietet keines dieser Projekte die Möglichkeit, Bedingungen über alle Teile einer ClassAd hinweg festzulegen. Gangmatching unterstützt das multilaterale Matchmaking durch den Einsatz von verschachtelten ClassAds (Ports) und kann mit beliebigen Typen von ClassAds

umgehen. Stork ist ein Broker, der ausschließlich auf Datendienste ausgelegt ist. Zum Planen von Jobs, die sowohl Datentransfers als auch Berechnungen enthalten, ist daher ein übergeordneter Scheduler wie der DAGMan nötig. Gangmatching und DAGMan bieten aber keine Möglichkeit, globale Bedingungen festzulegen. Sie eignen sich daher nicht für die oben erwähnten Optimierungen. SAMGrid ist ein Beispiel für den Einsatz von benutzerdefinierten Funktionen in ClassAds sowie für den Zugriff auf Dienste außerhalb des Matchmakers während des Matchmakings. Leider basiert das Projekt auf der alten ClassAd-Version. Darüber hinaus werden Jobs immer auf dem Rechner ausgeführt, der die meisten der benötigten Dateien vorrätig hat. Dies ist aber ungünstig, da die Warteschlangen der lokalen Zeitplaner von Ressourcen, deren Daten von vielen Jobs benötigt werden, schnell wachsen können [6]. Im Folgenden wird daher ein Resourcebroker vorgestellt, welcher unter anderem globale Bedingungen unterstützt. Einige interessante Aspekte der in Kapitel 7 vorgestellten Projekte werden in dieses Modell integriert.

8.2 Erweiterungen des Datenmanagementsystems

Der Resourcebroker soll in der Lage sein, das Replikat einer Datei auszuwählen, das am schnellsten bzw. kostengünstigsten zu einer bestimmten Ressource übertragen werden kann. Bei dieser Auswahl soll er durch das Datenmanagementsystem unterstützt werden. Zusätzlich soll der Broker das ausgewählte Replikat zu einer bestimmten Ressource übertragen lassen können. Für den Datentransfer bietet das ZIBDMS schon eine entsprechende Funktion im API. **`initSchedTransfer`** überträgt eine Datei in einem festgelegten Zeitrahmen von einer Ressource zu einer anderen.

Zur Auswahl des besten Replikats muss das ZIBDMS-API um eine Funktion **`GetBestSource`** erweitert werden. Diese benötigt Informationen über die Verteilung der Daten, die das Replikatmanagement des DMS zur Verfügung stellt sowie Informationen über die Netzwerkverbindungen, um Transferzeiten abschätzen zu können. Diese können bei einem Netzwerkinformationsdienst wie Delphoi abgefragt werden.

Die Aufgaben der Funktion **`getBestSource`** müssen nicht zwangsläufig in das DMS integriert werden, sondern könnten auch durch den Resourcebroker selbst durchgeführt werden. Für das DMS spricht jedoch, dass es für die Replikatverwaltung schon mit Delphoi kommuniziert und dafür ein eigenes API bereitstellt. So entfällt eine spezielle Schnittstelle des Brokers zu Delphoi oder einem anderen Informationsdienst. Zudem gehören die Bereiche Datenzugriff und Datentransfer konzeptionell eher zum Datenmanagementsystem als zum Broker, da dieser nicht primär für die Informationssammlung sondern für deren Auswertung verantwortlich ist. Nachteilig an dieser Vorgehensweise ist, dass die Nutzung ande-

rer Datenmanagementsysteme erschwert wird, da diese erst um die entsprechende Funktionalität erweitert werden müssen.

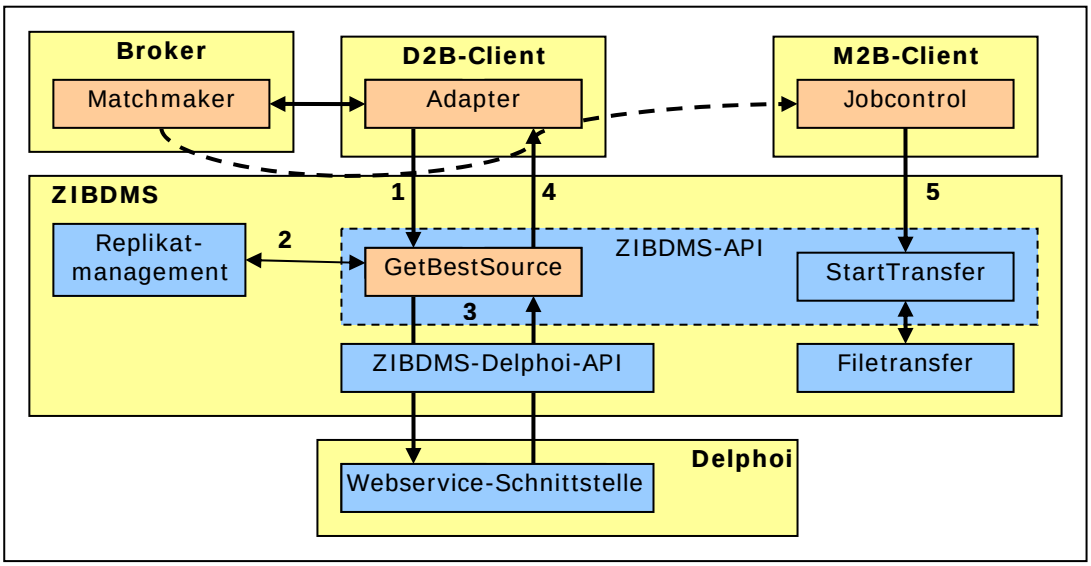


Abb. 27 Schema der Erweiterung des ZIBDMS
Anfrage nach dem besten Replikat einer Datei (1). Erfragen einer Liste aller Replikate (2). Übertragungszeit für jedes Replikat schätzen lassen (3). Zurücksenden des besten Replikats und der Übertragungszeit (4). Anforderung eines Datentransfers des Replikats (5). Orangefarbene Komponenten werden in dieser Arbeit entwickelt, blaue sind schon implementiert.

Der Ablauf der DMS-Abfrage durch den Broker ist in Abb. 27 dargestellt. Während des Matchmaking kann der Matchmaker durch eine benutzerdefinierte Funktion (siehe auch Abschnitt 9.2.4) Informationen über die Dateien eines DMS einholen. Dazu wird zunächst eine Anfrage an den zuständigen D2B-Client gerichtet, der diese weiter an das ZIBDMS-API gibt. Zusätzlich zu dem Dateinamen werden das Ziel eines möglichen Datentransfers sowie die Startzeit für diesen mit angegeben. Die API-Funktion g**etBestSource** erfragt beim Replikatmanagement eine Liste aller Replikate der Datei. Beim ZIBDMS heißt das, dass alle NSLs zurückgegeben werden, die die gleiche UFI wie die übergebene HFN besitzen. Für jedes dieser Replikate wird beim Netzwerkinformationsdienst die geschätzte Transferzeit zur angegeben Ressource angefragt. Für den Fall, dass die Anzahl der Replikate groß ist und das Replikatmanagement eine vorsortierte Liste zurückgibt, genügt es, nur einen Teil der Replikate prüfen zu lassen. „Vorsortiert“ bedeutet hier, dass die Replikate nach der durchschnittlich zur Verfügung stehenden Bandbreite geordnet sind. **getBestSource** gibt anschließend die Adresse des Replikats mit der kürzesten Übertragungszeit zurück. Zusätzlich berechnet die Funktion aus dem Startzeitpunkt und der geschätzten Dauer des Transfers die Kosten für diesen.

Da diese Information im Falle eines Matches mit in die MatchedClassAd übernommen wird, kann der für die Jobausführung verantwortliche M2B-Client den

Transfer des gewählten Replikats beim ZIBDMS-API über die Funktion **`initSchedTransfer`** veranlassen.

8.3 Der Resourcebroker

Der Resourcebroker (siehe Abb. 26) baut auf einer Client/Server-Architektur auf. Ähnlich wie bei Condor gibt es einen Pool aus Rechnern mit einer zentralen Instanz, dem ***Matchmaker***. Seine Hauptaufgabe ist, das Zusammenführen (Matchen) von passenden ClassAds. Für einen Pool wird immer genau ein Matchmaker benötigt, der auf einer ausgewählten Ressource ausgeführt wird und den anderen Komponenten mitgeteilt werden muss.

Neben diesem gibt es noch drei weitere Komponenten, die so genannten Clients, die als Schnittstelle zwischen einem Benutzer, einem Rechner oder einem DMS auf der einen Seite und dem Broker auf der anderen Seite dienen. Zusätzlich können die Clients auch bedingt untereinander kommunizieren, wobei diese dann teilweise selbst als Server fungieren. So hat der Matchmaker nach einem erfolgreichen Match nur die Aufgabe, den Auftraggeber zu informieren. Die anschließende Jobausführung einschließlich der Datentransfers wird dann selbstständig unter den Clients geregelt.

Der ***Job-to-Broker-Client*** (J2B-Client) ist die Schnittstelle zwischen dem Anwender, der einen Job ausführen möchte und dem Resourcebroker. Zusätzlich kann er u.a mit dem M2B-Client Dateien austauschen. Der J2B-Client muss auf allen Rechnern gestartet werden, von denen Jobaufträge abgeschickt werden sollen.

Der ***Data-to-Broker-Client*** (D2B-Client) erlaubt die Bekanntmachung eines Datenmanagementsystems im Pool und das Abrufen von Dateien und Dateiinformationen von einem solchen System. Für jedes DMS, welches Daten zur Verfügung stellen soll, wird eine Instanz des D2B-Clients benötigt.

Zwischen einem Rechner (*Machine*), der einen Job ausführen soll, und dem Broker ist der ***Machine-to-Broker-Client*** (M2B-Client) platziert, der zum einen den Rechner im Pool bekanntmacht und zum anderen für die lokale Jobausführung auf diesem Rechner verantwortlich ist. Auf allen Rechnern, die Jobs ausführen sollen, muss der M2B-Client gestartet werden.

Diese vier Komponenten setzen sich wiederum aus Komponenten zusammen, welche meist ähnliche Aufgaben übernehmen wie die Condor-Dämonen [13]. Tab. 5 listet diese Komponenten zusammen mit den vergleichbaren Dämonen auf. Eine Beschreibung der einzelnen Komponenten erfolgt in den folgenden Abschnitten.

Client	Dienst	Funktion	Condor-Dämon
-	-	Überwachung der anderen Dämonen	condor_master
M2B	M2B-provider	Repräsentation einer Ressource	condor_startd
M2B	M2B-jobcontrol	Ausführung und Überwachung von Jobs	condor_starter
M2B	M2B-scheduler	Ressourcenreservierung	-
J2B	J2B-provider	Jobübermittlung	condor_schedd
J2B	J2B-manager	Kommunikationskomponente zwischen Job und Rechner	condor_shadow
-	-	Informationssammlung über die anderen Condor-Dämonen	condor_collector
MM	mm	Matchmaker	condor_negotiator
-	-	Sensor für Tastaturaktivität	condor_kbdd
-	-	Checkpointserver	condor_ckpt_server
D2B	D2B-provider	Repräsentation eines DMS	-
D2B	D2B-adapter	Zugriffskomponente zum DMS	-

Tab. 5 Übersicht der einzelnen Brokerdienste mit ihren vergleichbaren Condor-Dämonen

8.3.1 Der D2B-Client

Der D2B-Client ist die Schnittstelle zwischen einem Datenmanagementsystem und dem Broker. Im Prinzip können durch Anpassung des Clients verschiedene Datenmanagementsysteme benutzt werden. Diese Arbeit beschränkt sich auf eine Kooperation mit dem ZIBDMS. Ein anderes DMS kann benutzt werden, wenn es Informationen über die Verteilung der Daten geben sowie Transferzeiten abschätzen kann.

Der D2B-Client ist aus zwei Objekten aufgebaut. Ein *Provider* ist dafür verantwortlich, das DMS im Pool bekanntzumachen. Dazu wird periodisch eine ClassAd erstellt und an den Matchmaker geschickt. DataClassAds haben ebenso wie MachineClassAds nur eine beschränkte Lebenszeit, z.B. 10 Minuten. Wird in dieser Zeit die ClassAd nicht erneuert, so nimmt der Matchmaker an, dass die entsprechende Ressource nicht mehr zur Verfügung steht und entfernt diese aus seiner Liste. Dies soll zum einen sicherstellen, dass der Matchmaker möglichst aktuelle Informationen besitzt, zum anderen soll die Wartezeit bei der Abfrage externer Systeme minimiert werden, um die Effizienz des Matchmakers zu erhöhen. Befinden sich diese Systeme auf anderen Ressourcen, so dauert aufgrund der nötigen Netzwerkkommunikation eine Anfrage wesentlich länger. Werden zudem Datenspeicher abgefragt, die nicht mehr verfügbar sind, würde die Wartezeit weiter ansteigen.

Das zweite Objekt, der *Adapter*, nimmt Anfragen vom Matchmaker und den einzelnen Clients an und wandelt diese, damit sie mit dem jeweiligen DMS kompatibel sind. Der Matchmaker richtet an dieses Objekt Anfragen zur Verfügbarkeit von Dateien und fordert eine Abschätzung für Dauer und Kosten eines möglichen

Datentransfers an. Der M2B-Client veranlasst notwendige Dateitransfers über den Adapter.

Eine graphische Übersicht über die Komponenten dieses und der anderen Clients findet sich im Abschnitt 8.3.5.

8.3.2 Der M2B-Client

Als Schnittstelle zwischen einer Ressource für die Jobausführung und dem Broker fungiert der M2B-Client. Ein *Provider* generiert periodisch eine MachineClassAd mit Informationen über die Ressource, auf der der Client ausgeführt wird und sendet diese an den Matchmaker. Diese Informationen sind zum einen statisch, etwa Angaben über den vorhandenen Arbeitsspeicher oder die Leistung der CPU als auch dynamisch, etwa die durchschnittliche Auslastung von CPU und Speicher.

Der *Scheduler* dient zum einen als Auskunftssystem für den Matchmaker und stellt Informationen zur Verfügung, ob die Ressource zu einer bestimmten Zeit verfügbar ist. Sofern andere Dienste auf dieser Ressource laufen, kann natürlich nur im Rahmen der schon geplanten Jobs Auskunft darüber gegeben werden. In diesem Zusammenhang unterstützt der Scheduler auch einen Reservationsservice, um eine Ressource für eine bestimmte Zeitspanne für einen Job zu reservieren. Der Resourcebroker dieser Arbeit kann damit direkt Einfluss auf die zeitliche Platzierung eines Jobs auf einer bestimmten Ressourcen nehmen, wie es bei anderen Brokern eher nicht üblich ist [36].

Die zweite Aufgabe des Schedulers ist, Jobs in einem Zeitplan zu verwalten und diese zu starten, sofern ihre Startzeit erreicht ist. Dazu wird vom Scheduler für jeden Job zunächst ein *JobControl*-Objekt erstellt welche mit dem aufrufenden Manager des J2B-Clients sowie den D2B-Clients kommuniziert, um benötigte Informationen und Daten für die Jobausführung zu erhalten. Anschließend ist das JobControl des M2B-Clients für die Ausführung des jeweiligen Programms sowie den Rücktransport der Daten verantwortlich. Je Client kann es zu einem Zeitpunkt mehrere JobControl-Objekte geben, die Auftrag, den Job auszuführen darf aber nur maximal ein solches Objekt besitzen.

Die Ausführung eines Jobs erfolgt im Batchbetrieb, d.h. der Auftraggeber hat keinen Einfluss auf die eigentliche Jobausführung. Benötigt ein Programm Eingaben, so müssen diese in einer Datei vorliegen, die dem Programm als Eingabestrom zur Verfügung gestellt wird. Der Auftraggeber kann lediglich den Job vorzeitig abbrechen.

8.3.3 Der J2B-Client

Der J2B-Client ist für den Anwender die Schnittstelle zum gesamten Grid. Übergibt dieser dem Client eine Jobbeschreibung, so erstellt der *Provider* daraus eine JobClassAd und sendet diese an den Matchmaker. Konnte ein Match realisiert werden, so wird der Provider vom Matchmaker benachrichtigt und erstellt ein *Manager*-Objekt. Dieses dient zum einen als Kontrollinstanz für die weitere Ausführung und zum anderen als Kommunikationspartner für das zugehörige JobControl-Objekt des M2B-Clients. Das Manager-Objekt sendet an den Scheduler des M2B-Clients den Auftrag, die Ausführung zu starten. Dadurch wird der Job, wie oben beschrieben, in die Warteschlange eingereiht und alle Vorbereitungen zur Ausführung des Jobs getroffen. Darüber hinaus besitzt der Client auch eine Jobverwaltung, mit Hilfe derer sich der Anwender jederzeit über den aktuellen Status seiner Jobs informieren kann. Andere Komponenten können dem J2B-Client dazu Statusänderungen eines Jobs mitteilen.

8.3.4 Der Matchmaker

Der Matchmaker ist die zentrale Instanz in einem Pool. Aus Gründen der Skalierbarkeit ist dies nicht als unbedingt optimal anzusehen. Zudem stellt es einen Single-Point-of-Failure dar. Dieses Problem kann dadurch umgegangen werden, dass ein Matchmaker nur für einen Teil der verfügbaren Rechner verantwortlich ist. Fällt dann dieser aus, so ist nur ein Teil der Ressourcen nicht mehr zugänglich, das System an sich funktioniert weiter. Ein Anwender kann seine Anfragen dann an einen anderen Matchmaker schicken, sollte der eigentliche Matchmaker nicht mehr verfügbar sein. Auch eine Kommunikation der Matchmaker untereinander wäre denkbar, damit diese beispielsweise die ihnen verfügbaren Ressourcen gemeinsam verwalten könnten oder damit ein Matchmaker nicht gematchte JobsClassAds an einen anderen übergeben könnte, um die Chance für einen erfolgreichen Match zu erhöhen.

Wie aus den obigen Abschnitten hervorgeht, erhält der Matchmaker von allen Clients zunächst einmal deren ClassAds, die in entsprechenden Listen gesammelt werden. Der Matchmaker versucht fortlaufend, nicht gematchte ClassAds miteinander zu matchen. Unter „matchen“ bzw. „Matchmaking“ versteht man, dass die **REQUIREMENTS**-Attribute der einzelnen ClassAds den Wahrheitswert „wahr“ ergeben müssen.

Beim Matchen müssen

- eine JobClassAd,
- eine MachineClassAd sowie
- eine oder mehere DataClassAds,

die zueinander passen, gefunden werden. Je Data- und MachineClassAd gibt es genau ein **REQUIREMENTS**-Attribut und ein **RANK**-Attribut, welche Anforderungen an den Job stellen können. Bedingungen zwischen Data- und MachineClassAds sind denkbar, in diesem Matchmaker aber nicht realisiert. In einer JobClassAd gibt es mindestens je zwei solche Attribute (siehe Abb. 28). Das eine gehört zu der inneren ClassAd **MACHINE** und beschreibt die Anforderungen, die der Job an die Ressource stellt. Diese können direkt durch den Benutzer festgelegt werden. Dies gilt ebenfalls für das **RANK**-Attribut im **MACHINE**-Teil. Zusätzlich gibt es in jeder JobClassAd noch ein globales **REQUIREMENTS**- und ein globales **RANK**-Attribut. Die globalen Attribute dienen dazu, Anforderungen festzulegen, die alle Teile der ClassAd zusammen betreffen. Beispielsweise können mit Hilfe dieser Attribute die Gesamtkosten des Jobs begrenzt werden.

Im Folgenden wird der Ablauf des Matchmakings beschrieben. Der Matchmaker durchläuft dazu die Liste der JobClassAds und versucht die einzelnen JobClassAds mit Machine- und DataClassAds zu matchen. Um die Auslastung des Rechners, der den Matchmaker repräsentiert, zu senken, wird ein Durchlauf nur gestartet,

- wenn es JobClassAds gibt, für die es bisher keinen Matchversuch gab,
- wenn eine neue Machine- oder DataClassAd zum Matchmaker übertragen wurde oder
- wenn eine Machine- oder DataClassAd aktualisiert wurde.

```
[
  EXECUTABLE = sample.class;
  UNIVERSE = java;
  MACHINE = [
    REQUIREMENTS = MACHINE.TYPE == „machine“ && MACHINE.MEMORY > 100;
    RANK = 1 / MACHINE.MEMORY;
  ]
  DATACOUNT = 1;
  DATAFILES = {
    [
      LABEL = „DATAFILE0;
      FILE = getFile(„sample.class“);
      REQUIREMENTS = DATAFILE0.TYPE == „data“ && FILE.ISAVAILABLE == true;
      RANK = 0;
    ]
  }
  MINTIME = true;
  REQUIREMENTS = 1/(MACHINE.Endtime - minimum(DATAFILES[0]))
  RANK = 0;
]
```

Abb. 28 Vereinfachtes Beispiel für eine JobClassAd
Die ClassAd besitzt drei REQUIREMENTS-Attribute, eines im Machine-Teil, eines in der Beschreibung für eine benötigte Datei sowie ein globales Attribut in der vorletzten Zeile. **getFile** ist eine benutzerdefinierte Funktion.

Abb. 29 zeigt den verwendeten Algorithmus in Pseudocode. Zunächst wird für jede Kombination aus JobClassAd und MachineClassAd geprüft, ob diese matchen können, d.h., ob die **REQUIREMENTS**-Attribute der MachineClassAd und des Machi-

ne-Teils der JobClassAd (**j.MACHINE**) den Wahrheitswert „wahr“ ergeben (Zeilen 3-5). In diesem Fall wird aus beiden ClassAds eine neue ClassAd **x** erstellt, die die beiden einzelnen ClassAds enthält. Zusätzlich werden noch Referenzen auf die jeweiligen ClassAds eingefügt (6). Benötigt ein Job keine Eingabedaten zur Ausführung und werden die globalen Anforderungen erfüllt, wurde ein Match gefunden (8) und es muss der Rang (**Rank**-Attribut) für die gesamte JobClassAd und der Rang für den Machine-Teil der JobClassAd berechnet werden. Beide Angaben dienen dazu, bei mehr als einem möglichen Match den besten auszuwählen. Die Rangwerte werden zusammen mit der MatchedClassAd **x** in einer Liste **mlist** gespeichert (9-11).

```
01 mlist ← {} // Liste der möglichen MatchedClassAds
02
03 foreach JobClassAd j
04   foreach MachineClassAd m
05     if j.MACHINE.REQUIREMENTS==true && m.REQUIREMENTS==true then
06       x ← merge(j, m)
07       if j.DATACOUNT==0 then
08         if x.REQUIREMENTS==true then
09           r0 ← x.j.RANK
10           r1 ← x.j.MACHINE.RANK
11           mlist.add(<x, r0, r1>)
12         endif
13       else
14         dlist ← {}
15         findData(x, 0)
16       endif
17     endif
18   endfor
19   if |mlist| > 0 then
20     mlist.sort(r0, r1)
21     startJob(mlist[0])
22   endif
23   mlist ← {}
24 endfor
25
26 findData(MatchedClassAd x, n) { // n=number of placed datafiles
27   jd ← x.j.DATAFILES[n]
28   foreach DataClassAd d
29     if !dlist.contains(<n,d>) && jd.REQUIREMENTS==true
          && d.REQUIREMTENS==true then
30       x ← merge(x, d)
31       if n+1 < |x.j.DATAFILES| then
32         findData(x, n+1)
33       else
34         if x.j.REQUIREMENTS==true // global requirements
35           r0 ← x.j.RANK
36           r1 ← x.j.MACHINE.RANK
37           mlist.add(<x, r0, r1>)
38         endif
39       endif
40       x ← remove(x, d)
41     else
42       dlist.add(<n,d>)
43     endif
44   endfor
45 }
```

Abb. 29 Algorithmus des Matchmaking im Resourcebroker

Für den Fall, dass eine oder mehr Dateien für den Job benötigt werden, wird die Funktion **finddata** gerufen, welche in **mlist** alle möglichen Matches zwischen

der JobClassAd, der MachineClassAds und den verfügbaren DataClassAds einträgt. Die Funktion wird weiter unten ausführlich beschrieben. (15).

Die Liste **mlist** enthält anschließend alle möglichen Matches für die aktuelle JobClassAd in Form von MatchedClassAds. Abb. 30 zeigt ein Beispiel für eine solche ClassAd, in die alle Original-ClassAds übernommen wurden. Zusätzlich wurden sie um Attribute erweitert, die sicherstellen, dass Referenzen auf die einzelnen ClassAds weiterhin gültig sind. Anschließend wird diese Liste nach den Rangwerten der gesamten JobClassAd und nach den Rank-Werten des Machine-Teils der JobClassAd sortiert. An der ersten Position der Liste befindet sich der Match, mit den höchsten Rangwerten (19-20). Zu beachten ist, dass jede JobClassAd mindestens zwei **RANK**-Attribute hat und jede DataClassAd und jede MachineClassAd ein weiteres.

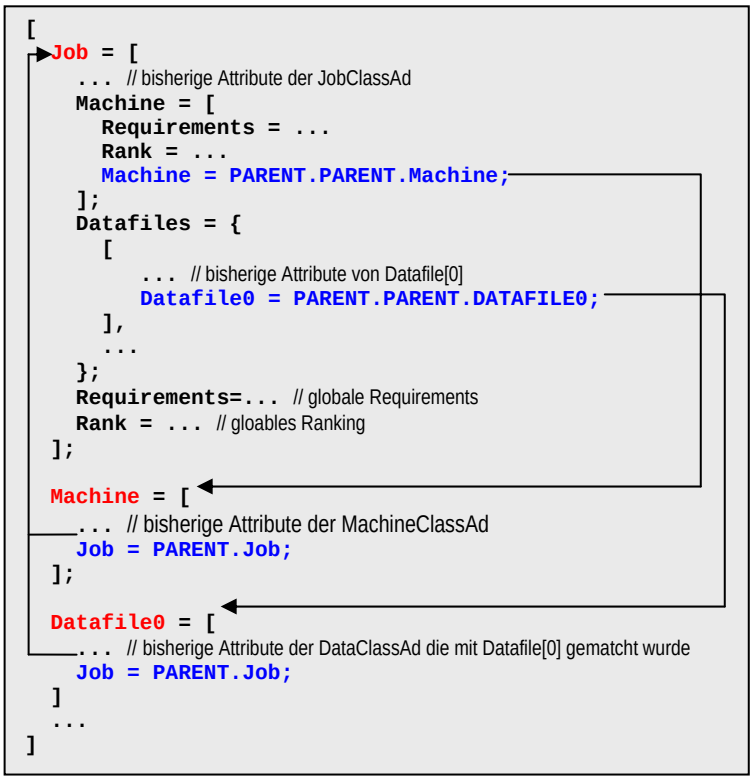


Abb. 30 Beispiel für eine MatchedClassAd
Die MatchedClassAd setzt sich aus den einzelnen ClassAds der Matches zusammen (rot gekennzeichnet). Um alle Attributreferenzen, die während des Matchmakings bestanden haben, zu erhalten, müssen zusätzliche Attribute eingefügt werden (blau). Die Pfeile zeigen die Ziele der Referenzen an.

Es ist nötig, zu klären, wie diese verschiedenen Rangwerte zu interpretieren sind, d.h., ob die einzelnen Rangwerte gleichberechtigt anzusehen sind oder ein Ranking höherwertig als ein anderes ist. Beispielsweise kann der Gesamtrang als Produkt aller anderen Rangwerte angesehen werden. Dies ist aber nicht unbedingt sinnvoll. Hauptziel des Brokers ist, den Job global zu optimieren, d.h. die Gesamtkosten- bzw. die Gesamtausführungszeit zu begrenzen oder zu minimie-

ren. Daher wird dem globalen Rang der JobClassAd die höchste Priorität zugeordnet. Für den Fall, dass es zwei MatchedClassAds mit dem gleichen globalen Job-Rang gibt, wird der Job-Machine-Rang (also **JOB.MACHINE.RANK**) ausgewertet. Die **Rank**-Attribute der DataClassAd und MachineClassAd werden hier ignoriert.

Die MatchedClassAd, die an der ersten Position der Liste **mlist** steht, wird an die Funktion **startJob** übergeben, die die Absender der ClassAds informiert und die zugehörige JobClassAd aus der Liste der JobClassAds löscht (21). Damit ist das Matchmaking der JobClassAd beendet und der Matchmaker versucht, die nächste JobClassAd der Liste zu matchen.

Das Suchen von passenden Datenquellen wurde in eine eigene Funktion **findData** ausgelagert, da die Suche rekursiv mittels Backtracking erfolgt. Zunächst wird für die erste Datei ein Datenspeicher gesucht, der ein Replikat dieser besitzt, dann ein Datenspeicher für die zweite usw. bis entweder für alle Dateien eine Quelle gefunden wurde oder keine Kombination möglich ist. Im ersten Fall ist ein Match zustande gekommen. In jedem Fall wird der zuletzt gewählte Datenspeicher wieder entfernt und durch einen noch nicht getesteten ersetzt, bis alle möglichen Kombinationen probiert wurden.

Der Ablauf im Einzelnen: Angenommen, es gibt einen Match aus JobClassAd und MachineClassAd **x** und zwei Dateien, für die Datenspeicher gesucht werden. Als erstes wird für die erste Datei ein Datenspeicher gesucht, indem für alle vorhanden DataClassAds geprüft wird, ob deren Anforderungen und die das DataFile-Teils der JobClassAd erfüllt sind (27-29). Ist dies für eine DataClassAd nicht der Fall, wird die nächste in der Liste geprüft. Ansonsten wird die DataClassAd in die bisherige MatchClassAd eingefügt (30). Für den Fall, dass noch nicht für alle Dateien ein Datenspeicher gefunden wurde, wird die Funktion **findData** nochmals mit einem um eins erhöhten **n** gerufen, welches angibt, dass jetzt für die zweite Datei ein Datenspeicher gesucht wird (31-32). Wird auch für die zweite Datei ein Datenspeicher gefunden, ist die Bedingung in (31) nicht mehr erfüllt. In diesem Fall sind für alle Dateien Datenspeicher gefunden worden. Sofern die globalen Anforderungen erfüllt sind werden die Rangwerte wie oben berechnet und zusammen mit der MatchedClassAd in der Liste **mlist** gespeichert (34-37). Anschließend wird die zuletzt eingefügte DataClassAd wieder entfernt, um die nächste DataClassAd zu testen (40).

Die Laufzeit des Algorithmus kann reduziert werden, indem für jede Datei die Datenspeicher vermerkt werden, die kein Replikat der Datei besitzen. Dazu wird in (14) eine Liste **dlist** definiert. Sind die Anforderungen von Datei und Datenspeicher nicht erfüllt, wird diese Kombination in die Liste mit aufgenommen (42). Testet der Algorithmus die gleiche Kombination nochmals, kann vorab geprüft

werden, ob **dlist** schon einen Eintrag besitzt, der besagt, dass die Datei nicht vorhanden ist. In diesem Fall muss der Datenspeicher nicht ein weiteres Mal geprüft werden.

8.3.5 Beispiel für eine Jobanfrage

Bevor eine Jobanfrage gematcht werden kann, müssen die Ressourcen und Datenspeicher des Grids beim Matchmaker bekannt gemacht werden. Die Bearbeitung eines Jobs erfolgt dann in mehreren Schritten, die hier noch mal zusammengefasst werden:

1. Erstellen und Bekanntmachen des Jobs
2. Matchmaking des Job mit passenden Ressourcen und Datenspeichern
3. Jobvorbereitung, u.a. Erstellen der Ausführungsumgebung, Datentransfers
4. Jobausführung
5. Bereinigung der Ressource, Zurücksenden der Ergebnisse

Im Folgenden wird dieser Ablauf ausführlicher beschrieben und in Abb. 31 bis Abb. 33 grafisch dargestellt.

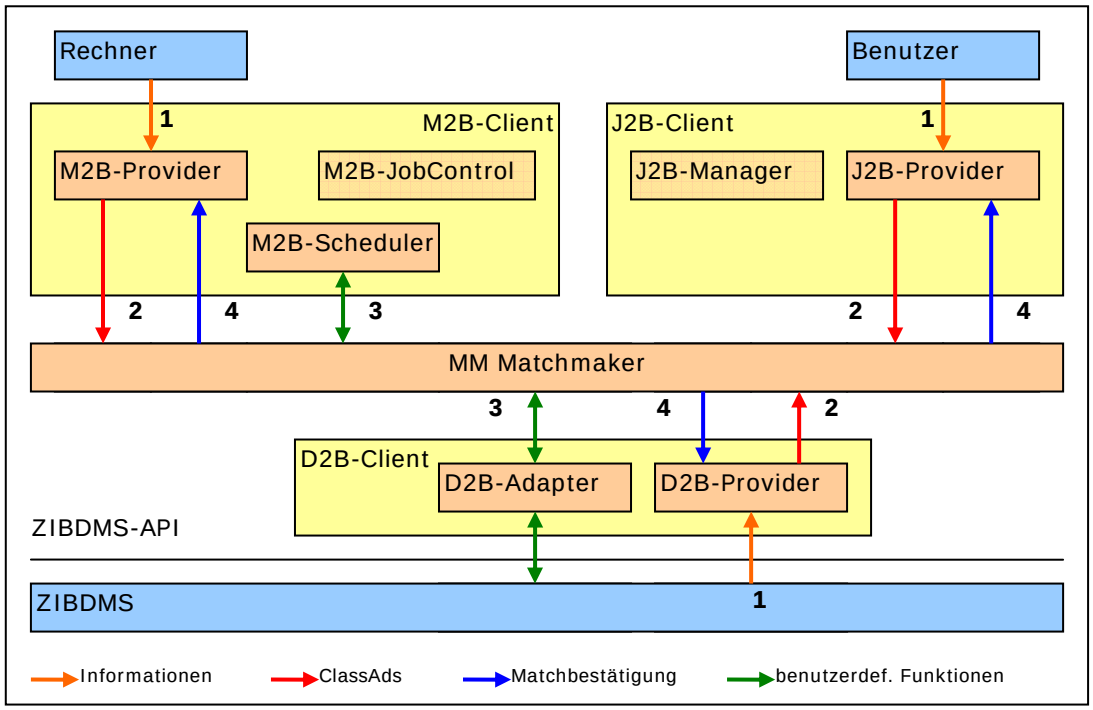


Abb. 31 Bekanntmachung und Matchmaking

Vorbereitung: Zunächst erstellen die Provider der D2B-Clients und der M2B-Clients je eine ClassAd, die Informationen über die zugrunde liegende Ressource enthält, und senden diese an den Matchmaker (1, 2).

***Joberstellung**:* Will der Benutzer einen Job in Auftrag geben, so erstellt er eine Beschreibung des Jobs (1). Diese wird dann vom J2B-Client an den Matchmaker geschickt (2). Jobbeschreibungen sehen JobClassAds recht ähnlich. Die Unterscheidung wurde vom Condor-System so übernommen, um möglichst kompatibel zu diesem zu bleiben. Vorteil der Jobbeschreibung ist, dass der Provider zum einen fehlende, aber benötigte Attribute mit Standardwerten einfügen kann und zum anderen den Anwender entlastet, indem bestimmte komplizierte Attributkonstrukte in ClassAds automatisch aus kurzen Anweisungen des Benutzers generiert werden können. Will der Benutzer beispielsweise eine maximale Gesamtausführungszeit für einen Job festlegen, muss er lediglich in der Jobbeschreibung ein entsprechendes Attribut einfügen, dessen Wert die entsprechende Zeitspanne angibt. Daraus erstellt der Provider die nötigten Formulierungen für das **`Requirements`**-Attribut der JobClassAd, in diesem Fall, dass die Zeitspanne vom Beginn des ersten Datentransfers bis zum Ende des eigentlichen Jobs kleiner sein muss als die angegebene Zeitspanne.

***Matchmaking*:** Der Matchmaker versucht, solange es ClassAds gibt, die noch nicht gematcht wurden, diese mit passenden ClassAds zusammenzuführen. Dabei müssen in jeder beteiligten ClassAd die **`REQUIREMENTS`**-Attribute erfüllt sein. Während des Matchens können über spezielle Schnittstellen noch zusätzlich Informationen von den D2B-Clients (Adapter) und den M2B-Clients (Scheduler) eingeholt werden (3). Dazu zählen z.B.:

- Ist der Rechner *m* von 5 bis 6 Uhr verfügbar?
- Hat das Datenmanagement *x* die Datei *y*?
- Wie lange braucht das DMS *x* die Datei *y* nach *m* zu transferieren?

Konnte ein Match realisiert werden, benachrichtigt der Matchmaker die betroffenen Clients und veranlasst eine temporäre Reservierung der Ressourcen (4). Dies verhindert, dass in der Zeit, in der der J2B-Provider über den Match informiert wird, die gleiche Ressource für die gleiche Zeitspanne für einen anderen Match eingeplant wird.

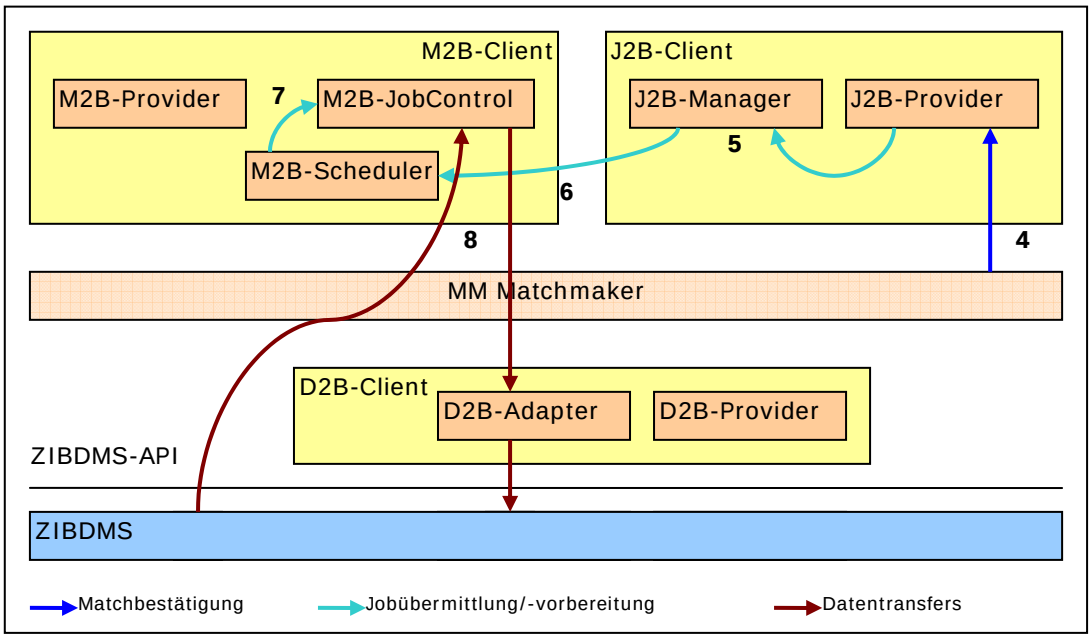


Abb. 32 Jobübermittlung und Datentransferanforderung

Jobvorbereitung und -ausführung: Der J2B-Client veranlasst den M2B-Client, den Job auf der gewählten Ressource auszuführen. Zunächst erstellt der J2B-Client dafür ein neues Manager-Objekt, welches sich um die Ausführung kümmert. Dies soll sicherstellen, dass der Provider nicht blockiert wird und sich wieder seiner Aufgabe, dem Entgegennehmen weiterer Jobaufträge, widmen kann (Abb. 32, Schritt 5). Der Manager kontaktiert den Scheduler des M2B-Clients, welcher den Job in seinen Zeitplan einordnet. Der Scheduler verwaltet lediglich die Jobs, d.h. er versucht nicht, die Ausführung der Jobs durch eine andere Reihenfolge zu optimieren. Er prüft den Zeitplan permanent um ggf. Jobs abzubrechen, die nicht in der reservierten Zeitspanne beendet werden konnten und um wartende Jobs zu starten (6). Durch den Scheduler wird für jeden Auftrag ein Jobcontrol-Objekt erstellt (7), welches vor der eigentlichen Jobausführung vom J2B-Client und vom D2B-Client die benötigten Dateien anfordert (8). Wurden die Vorbereitungen abgeschlossen und ist die Startzeit des Jobs erreicht bzw. warten keine anderen Jobs vor diesem, so wird das Jobcontrol mit der Ausführung des Jobs beauftragt (9). Es startet das jeweilige Programm und überwacht dessen Ausführung. Es informiert sowohl den Scheduler als auch den J2B-Manager über den Stand der Ausführung (10 und 11).

Bereinigung: Nach dem Programmende, veranlasst das Jobcontrol die Übertragung aller Ergbebnisdaten an den aufrufenden Client, wozu es mit dem zugehörigen Manager des J2B-Clients kommuniziert (12).

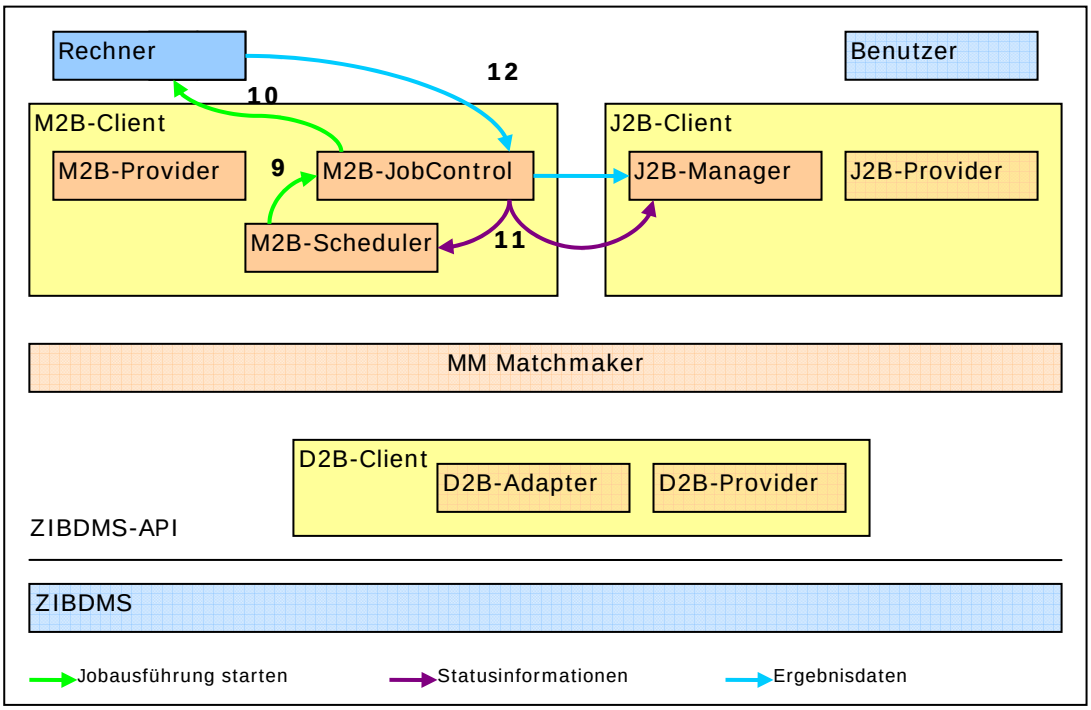


Abb. 33 Jobausführung und Zurücksenden der Ergebnisse

8.3.6 Vergleich des Resourcebrokers mit den 10 nötigen Aktionen für Grid Scheduling nach Schopf

In Abschnitt 2.1 wurden die Schritte vorgestellt, die für die Platzierung im Grid nach [38] notwendig sind. In diesem Abschnitt wird kurz beschrieben, wie diese dem Resourcebroker dieser Arbeit zugeordnet werden können.

Autorisierung: Wie oben erwähnt werden Autorisierung und Authentifizierung in dieser Arbeit vernachlässigt, daher entfällt in der aktuellen Version dieser Schritt.

Anforderungsdefinition: Jede beteiligte Komponente (Jobs, Ressourcen, DMSe) kann Anforderungen an die beteiligten Komponenten stellen, wobei diese sowohl statische als auch dynamische Eigenschaften mit einbeziehen können.

Ressourcenauswahl (Schritt 3-5): Zur Ausführung kommen nur Ressourcen, deren Anforderungen erfüllt sind, d.h. deren **REQUIREMENTS**-Attribut zum Wahrheitswert „wahr“ ausgewertet werden kann. Während des Matchmaking können auch dynamische Eigenschaften abgefragt werden.

Reservierung: Kommt ein Match zustande, so wird die beteiligte Ressource für eine festgelegte Zeitspanne reserviert. Diese Reservierung soll verhindern, dass die gleiche Ressource nochmals durch den Broker verplant wird.

Jobübermittlung: Im Falle eines Matches schickt der Matchmaker die gematchte ClassAd an den J2B-Client zurück. Dieser leitet dann den Job an die jeweilige Ressource zur Ausführung weiter.

Vorbereitung und Jobstart: Der M2B-Client veranlasst vor der Ausführung des Jobs alle nötigen Datentransfers. Anschließend startet der Client die Ausführung.

Überwachung: In der aktuellen Version wird auf eine Überwachung des Jobs weitestgehend verzichtet. Lediglich das Ende des Prozesses wird abgewartet.

Aufräumen: Nach erfolgreichem Ende des Jobs schickt der M2B-Client alle Ergebnisdateien, sofern dies durch den Nutzer gewünscht wurde, an diesen zurück. In jedem Fall werden alle Dateien, die durch oder für den Job angelegt wurden, gelöscht.

9 Implementation des Resourcebrokers

Diese Kapitel soll beschreiben, wie die im vorherigen Kapitel vorgestellte Architektur implementiert wurde und erklären, warum bestimmte Mechanismen wie Threads benutzt wurden oder auf andere Technologien wie Webservices verzichtet wurde. Zunächst wird die Implementation der Erweiterungen am ZIBDMS beschrieben, anschließend die Implementation des Resourcebrokers.

9.1 ZIBDMS-Erweiterungen

Im Rahmen der erwähnten Diplomarbeit von Monika Moser wurde parallel zu dieser Arbeit ein API zum Zugriff auf einen Delphoi-Server aus dem ZIBDMS heraus implementiert. Für eine Zusammenarbeit von Resourcebroker und ZIBDMS musste daher das ZIBDMS-API nur um eine Funktion **`getBestSource`** erweitert und eine entsprechende CORBA-Schnittstelle erstellt werden. Die Funktion ermittelt für den übergebenen Dateinamen (in Form eines HFNs) über **`ReplicaLoction->getLocations`** alle Replikate dieser Datei und übergibt diese zusammen mit der Dateigröße und dem Transferziel an den Delphoi-Server, der für jedes Replikat die geschätzte Transferzeit in Sekunden ermittelt. Anschließend sortiert die Funktion die Liste der Replikate anhand der Zeiten und gibt das Replikat mit der geringsten Ausführungszeit zurück.

9.2 Der Resourcebroker

Der Resourcebroker wurde in der Programmiersprache Java implementiert, da diese plattformunabhängig ist, was bei Grids aufgrund ihrer heterogenen Architektur von Vorteil ist. Der modulare Aufbau des Systems mit einer zentralen Komponente und den einzelnen Clients wurde bei der Implementation beibehalten. Er entspricht der Client/Server-Architektur wobei die Clients zumindest zeitweise auch als Server gegenüber den anderen Clients fungieren können. Im folgenden Abschnitt wird zunächst das Kommunikationsprotokoll zwischen den einzelnen Komponenten beschrieben.

9.2.1 Remote Method Invocation

Zur Kommunikation untereinander nutzen die einzelnen Komponenten *Remote Method Invocation* (RMI) [24]. Webservices wurden nicht benuzt, da die Ausführungsgeschwindigkeit von Webserviceanfragen gegenüber RMI-Anfragen wesentlich höher liegt [14]. Eine spätere Erweiterung durch Webservices, die auf der RMI-Schnittstelle aufbauen, ist jedoch denkbar.

RMI ist seit Version 1.1 im JDK[47] enthalten und erlaubt, Objekte im Netz zu verteilen und so Anwendungen auf anderen Rechnern zur Verfügung zu stellen. Im Folgenden wird die Funktionsweise kurz beschrieben (siehe Abb. 34):

1. Zunächst werden in einem *Remote-Interface* (abgeleitet vom Interface Remote) die Funktionen definiert, die verteilt zugänglich sein sollen.
2. Anschließend wird eine Serverklasse erstellt, die das obige Interface implementiert. Daraus erzeugte Objekte werden *Remote-Objekte* genannt.
3. Um Auskünfte an Clients über verfügbare Dienste geben zu können, existiert auf jedem Rechner, der als Server fungiert, ein *Naming-Service*. Für diese Arbeit wurde der im RMI-Paket enthaltene Dienst *RMI-Registry* verwendet.
4. Mit Hilfe der RMI-Registry erhalten die Clients Referenzen (so genannte *Remote-Referenzen*) auf die angefragten Objekte und können dort die gewünschten Methoden aufrufen. Werden beim Aufruf Parameter angegeben, so werden diese ebenfalls an das Remote-Objekt übertragen. Der Rückgabewert der aufgerufenen Methode wird anschließend an den Client zurückgesendet.

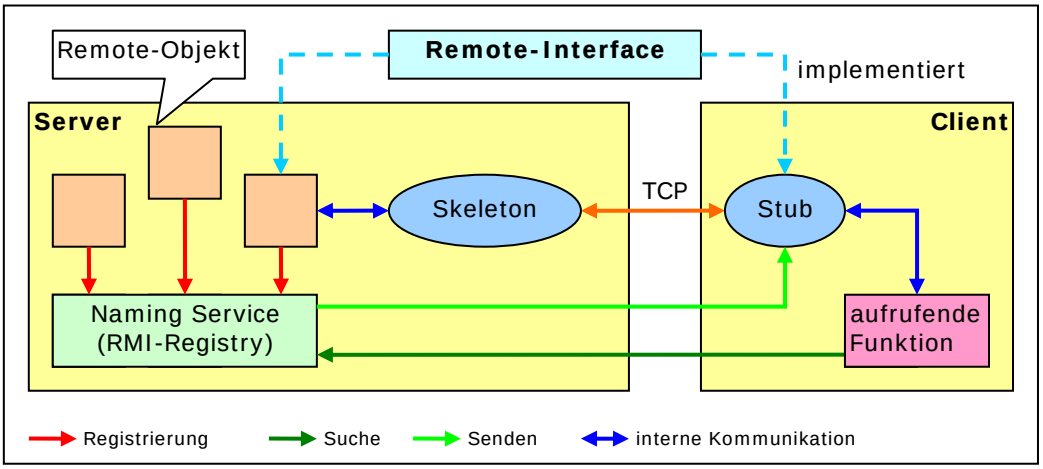


Abb. 34 Arbeitsweise von RMI, nach [24]

Der Client kommuniziert jedoch nicht direkt mit dem serverseitig gespeicherten Objekt. Stattdessen erhält der Client von der RMI-Registry ein Objekt einer Stub-Klasse, die ebenfalls das Remote-Interface implementiert. Das Objekt dient als Platzhalter für das eigentliche Remote-Objekt. Der Stub kontaktiert das Gegenstück auf dem Server, das sogenannte Skeleton, via TCP-Verbindung. Dem Gegenstück ist das tatsächliche Remote-Objekt bekannt, so dass es Anfragen vom Stub an dieses weiterleiten kann. Rückgabewerte werden dann entspre-

[47] *Java Development Kit*

chend vom Objekt über das Skeleton zurück zum Stub übertragen. Ziel der Kommunikation der RMI ist, verteilte Zugriffe möglichst transparent zu gestalten.

Die Implementierung unterscheidet sich von der Architektur des Resourcebrokers vor allem in genau diesen RMI-Komponenten. Im Gegensatz zum in Kapitel 8 vorgestellten Modell kommunizieren die Objekte der einzelnen Komponenten des Resourcebrokers nicht direkt miteinander. Vielmehr besitzen dort alle Objekte ein Remote-Interface namens *Services* sowie eine Implementierung dessen in der Klasse ServicesImpl. Eine Anfrage einer Komponente wird per RMI zunächst an ein Objekt der Klasse ServicesImpl weitergeleitet. Dieses leitet dann die Anfrage an das zuständige Objekt weiter. Bei der Beschreibung und Bebilderung der einzelnen Komponenten wird der Name des Interfaces anstelle des Namens der Klasse verwendet um hervorzuheben, dass dem jeweiligen Client nur das Interface bekannt ist.

Sinn dieser zusätzlichen Klassen soll sein, die Programmlogik von der Kommunikation zu trennen. Um beispielsweise eine andere Kommunikationsmethode zu wählen, genügt es, die ServiceImpl-Klassen entsprechend anzupassen. Änderungen an den einzelnen Komponenten darüber hinaus sind nicht notwendig.

9.2.2 Jobbeschreibungen

Zunächst wird hier der Aufbau von Jobbeschreibungen erläutert, bevor im folgenden Abschnitt die Umwandlung derer in JobClassAds beschrieben wird. Jobbeschreibungen sind Textdateien, die je Zeile ein Attribut der Form „Bezeichner=Wert“ haben (siehe Anhang, Abschnitt 13.1). Diese Dateien enthalten alle Angaben und Anforderungen die für die Ausführung eines Jobs notwendig sind. Jobbeschreibungen sehen JobClassAds zwar recht ähnlich, sind mit ihnen jedoch nicht gleichzusetzen. Vorteil der Beschreibungen ist, dass der Benutzer nicht alle Standardattribute eintragen muss, da dies bei der Erstellung der JobClassAd automatisch passiert. Des Weiteren können komplizierte Ausdrücke in ClassAds aus kurzen Schlüsselwörtern in der Jobbeschreibung erstellt werden, was dem Nutzer viel Arbeit erspart.

Wenn möglich wurden die Bezeichner in der Jobbeschreibung so benannt wie in Condor, um die Kompatibilität zum Condor-System sicherzustellen. Einige Bezeichner werden jedoch in einer anderen Weise interpretiert. Ein Beispiel ist das Attribut **`TRANSFER_INPUT_FILES`**, welches Condor veranlasst, die angegebenen Dateien vor dem Job zur ausführenden Ressource zu übertragen. Bei Condor muss an dieser Stelle eine vollständige Pfadangabe für jede Datei angegeben werden. Dem Resourcebroker dieser Arbeit reichen hier logische Dateinamen, da

er vor der Jobausführung automatisch Datenspeicher sucht, die die benötigten Dateien besitzen.

Jede Jobbeschreibung muss die drei Bezeichner EXECUTABLE, UNIVERSE und QUEUE enthalten, deren Bedeutungen in der folgenden Tabelle zusammengefasst sind.

EXECUTABLE	Gibt den Dateinamen des auszuführenden Programms an.
UNIVERSE	Gibt an, um was für einen Typ es sich handelt. Condor unterscheidet insgesamt acht verschiedene Universen [13], über die die Ausführungsumgebungen der einzelnen Jobs festgelegt werden. Von diesen unterstützt der Resourcebroker in der aktuellen Implementaion nur **STANDARD** und **JAVA**, wobei der einzige Unterschied darin besteht, dass beim Aufruf im **UNIVERSE JAVA** vor das Programm 'java' gestellt wird und ein ggf. vorhandenes '.class' im Namen des Executables entfernt wird. Die anderen Universen beziehen sich auf spezielle Arten von Jobs, die in dieser Arbeit nicht behandelt werden.
QUEUE	Gibt an, dass eine JobClassAd erstellt werden soll.

Tab. 6 Übersicht der Bezeichner in Jobbeschreibungen, Teil 1

Ein weiterer Vorteil von Jobbeschreibungen ist, dass ein Benutzer mit einer Jobbeschreibung mehr als einen Job erstellen lassen kann, indem das Schlüsselwort **QUEUE** mehrfach benutzt wird. Will man beispielsweise einen Job, d.h. ein Programm mit den gleichen Eingabedaten, mehrfach hintereinander ausführen, wobei jeweils einige Parameter schrittweise geändert werden (*Parameter Sweep Study*), so muss nicht für jede Job-Parameter-Kombination eine eigene Beschreibung erstellen werden.

Sind Bezeichner mehrfach vorhanden, so gilt immer der letzte gelesene Wert vor einem **QUEUE**. Im Beispiel (siehe Anhang, Abschnitt 13.1) bekommt die erste ClassAd als **INTIAL_DIR** 'run_1' zugeordnet. Vor dem zweiten **QUEUE** wird dann **INITIAL_DIR** durch den Wert 'run_2' überschrieben. Somit benutzt die zweite ClassAd diesen Wert dann als **INITIAL_DIR** und es konnten mit einer Jobbeschreibung zwei einzelne Jobs erstellt werden.

In der folgenden Tabelle werden alle weiteren Bezeichner aufgelistet, die vom J2B-Client besonders interpretiert werden. Alle anderen Bezeichner-Wert-Paare werden 1:1 in die JobClassAd übernommen.

PARAMETERS	Parameter, mit denen das Programm gestartet werden soll. Das aufzurufende Programm setzt sich wie folgt zusammen: **[java] EXECUTABLE [PARAMETERS] [< IN] [1> OUT] [2> ERR]**.
IN	(Logischer) Name einer Datei, die statt stdin benutzt wird.
OUT	Name einer Datei, die statt stdout benutzt wird. D.h. alle Ausgaben des Programms werden in dieser Datei gespeichert. Wird **OUT** angegeben, wird nach erfolgreicher Beendigung des Jobs diese Datei zurück an den aufrufenden Rechner kopiert.

ERR	Name einer Datei, die statt stderr benutzt wird. Diese Datei wird ebenfalls nach dem Jobende zurückgeschickt.
LOG	Name einer Datei, in die Informationen zur Jobausführung geschrieben werden, u.a. Anfangs- und Endzeit des Jobs. Sofern keine Umleitung für stderr angegeben wurde (siehe **ERR**), werden im Logfile auch alle Ausgaben auf stderr gespeichert. Wurde **LOG** in der Jobbeschreibung nicht angegeben, so wird **LOG=logfile** benutzt. Diese Datei wird ebenfalls an den Client zurückgeschickt.
REQUIREMENTS	Gibt die Anforderungen an den ausführenden Rechner an. Fehlt dieses Attribut, wird **REQUIREMENTS=true** gesetzt.
RANK	Gibt an, wie passende Rechner geordnet werden sollen. Fehlt die Angabe, wird **RANK=0** gesetzt.
DEADLINE	Eine Datums- und Zeitangabe, bis zu der der gesamte Job abgeschlossen sein muss.
MAXCOSTS	Maximalkosten des Jobs.
STARTTIME	Eine Datums- und Zeitangabe, zur der der eigentliche Job oder die Datentransfers starten (siehe **STARTTIMEFOR**). Fehlt die Angabe wird die aktuelle Zeit benutzt.
STARTTIMEFOR	Gibt an, ob **STARTTIME** die Anfangszeit für den eigentlichen Job sein soll (dann hat es den Wert **"JOB"**) oder ob es den Zeitpunkt des ersten Datentransfers angibt (jeder andere Wert). Fehlt die Angabe wird **STARTTIMEFOR="JOB"** benutzt.
MINCOSTS=true	Gibt an, dass die Kosten minimiert werden sollen.
MINTIME=true	Gibt an, dass die Gesamtausführungszeit minimiert werden soll.
TRANSFER_INPUT_FILES	Gibt eine Liste von Dateien an, die vor der Jobausführung zur Ressource kopiert werden müssen. Werden diese nicht auf dem lokalen System gefunden, werden sie bei Datenmanagementsystemen angefragt.
TRANSFER_OUTPUT_FILES	Gibt eine Liste von Dateien an, die nach der erfolgreichen Jobausführung von der Ressource zurück an den Client geschickt werden sollen.

Tab. 7 Übersicht der Bezeichner in Jobbeschreibungen, Teil 2

9.2.3 Der J2B-Client

Als Schnittstelle zwischen dem Jobauftrag und dem Resourcebroker dient der J2B-Client. Seine Aufgaben sind:

- die Annahme von Jobbeschreibungen,
- die Erstellung von JobClassAds aus Jobbeschreibungen und Weiterleitung dieser an den Matchmaker,
- die Informationssammlung und -auskunft zu allen angenommenen Jobbeschreibungen sowie
- die Annahme gematchter ClassAds und die Initialisierung der Ausführung.

Nach dem Start des J2B-Client werden zwei Threads gestartet. Der erste davon ist ***Joblist***, und verwaltet eine Liste aller Jobbeschreibungen, die vom Benutzer an den Client geschickt wurden. Gespeichert werden der aktuelle Zustand des Jobs sowie eine Zeichenkette, die zusätzlich zum Zustand noch ergänzende Bemerkungen erlaubt. Sofern ein Job bzw. dessen ClassAd an den Matchmaker übermittelt wurde, werden auch alle *ClassAdIds* zusätzlich zum Job gespeichert. Eine ClassAdId ist eine durch den Matchmaker vergebene eindeutige Identifikati-

onsnummer für die ClassAd, die durch alle weiteren Komponenten zur Zuordnung benutzt wird. Für jede ClassAd können ebenfalls ein Zustand und eine Bemerkung gespeichert werden. Objekte anderer Komponenten können per RMI den Status eines Job bzw. einer ClassAd jederzeit ändern.

Abb. 35 zeigt alle Zustände, die der Job (blaue Kreise) und die JobClassAds (blaue und orangefarbene Kreise) annehmen können. Der Zustand eines JobClassAd wird durch ein ein JobState-Objekt repräsentiert. Wird eine neue Jobbeschreibung angenommen wurde, wird ein JobState mit dem Zustand „Unknown“ erstellt. Hat der J2B-Provider aus der Beschreibung alle JobClassAds erstellt und an den Matchmaker geschickt, so wird der Status des JobStates auf „Created“ gesetzt. Im Folgenden ändert sich der Status der gesamten JobClassAd nicht mehr. Vielmehr wird für jede vom Matchmaker empfangene ClassAdId ein Untereintrag im JobState-Objekt angelegt, welches zunächst den Status „Waiting“ bekommt, bis die ClassAd gematcht wurde. Anschließend werden die Zustände „Placed“ (der Job wurde an eine Ressource übertragen), „Prepared“ (die Vorbereitungen auf der Ressource wurden abgeschlossen) und „Running“ (der Job wird ausgeführt) erreicht. Konnte der Job erfolgreich abgeschlossen werden und wurden alle Ergebnisdaten erfolgreich zurückgeschickt, so geht das JobState-Objekt für die jeweilige ClassAdId in den Zustand „Finished“ über, der dem Benutzer mitteilen soll, dass die Ergebnisdaten vorliegen. „Error“ kann vom jedem anderen Zustand erreicht werden, wenn ein Fehler auftritt. Beispiele für diese Fehler sind in der Abbildung vermerkt.

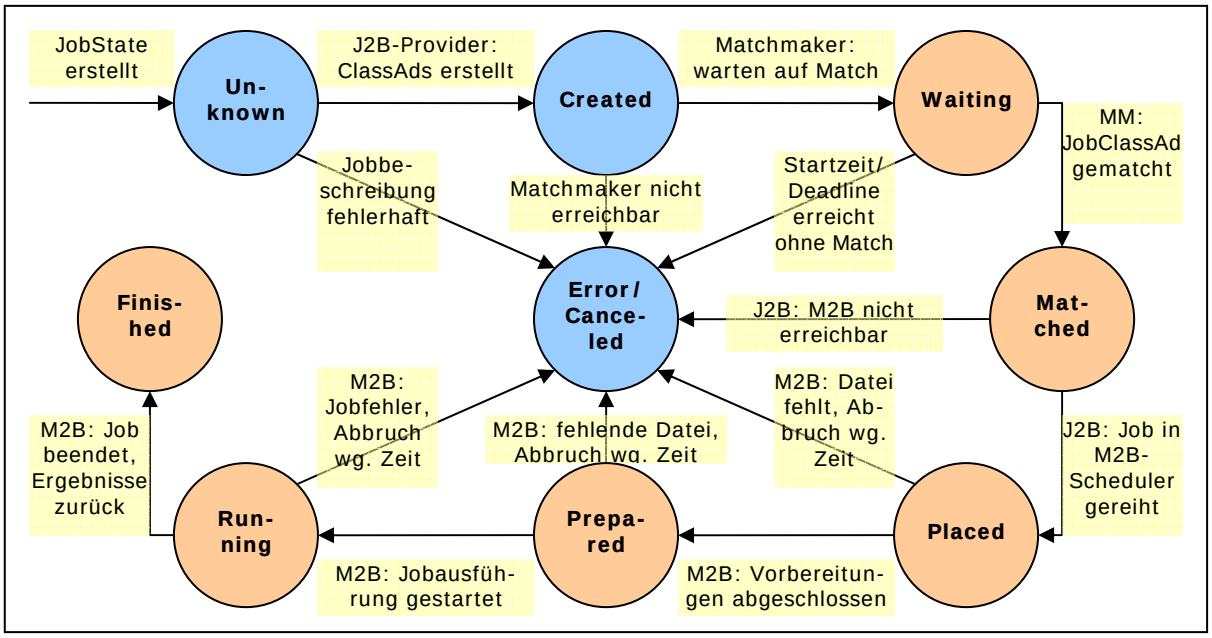


Abb. 35 Diagramm der Zustände eines JobState-Objekts

In der aktuellen Implementation dienen die Zustände nur der Information des Anwenders. Es ist jedoch denkbar, Kontrollmechanismen auf Basis der Zustände

einzurichten, welche bei einem Fehler diesen analysieren könnten und am zuletzt erreichten Zustand die Ausführung nach einer Fehlerkorrektur fortsetzen.

Der zweite Thread ist ein Objekt vom Typ ***Provider***. Der Provider ist zunächst dafür verantwortlich, eine eingehende Jobbeschreibung in eine bzw. mehrere ClassAds zu übersetzen.

Nach dem Start dieser zwei Threads wartet der Hauptthread auf Befehle des Anwenders. Alle Clients und der Matchmaker unterstützen die folgenden Befehle:

HELP zeigt eine Liste der verfügbaren Befehle an.

INFO zeigt spezifische Informationen zum Dienst an. Beim J2B-Client z.B. alle Jobs inkl. deren Status.

STOP beendet den jeweiligen Dienst.

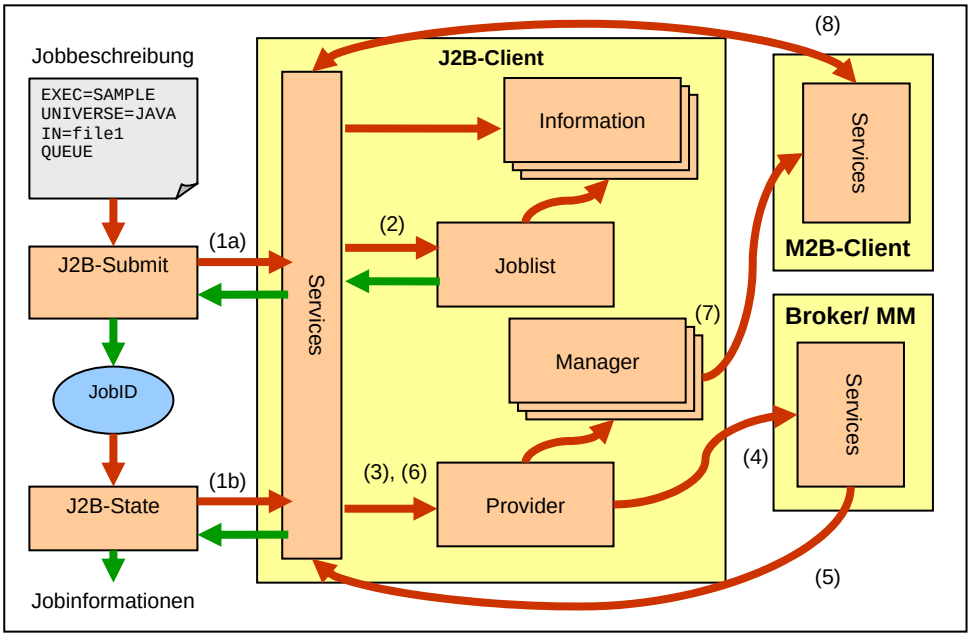


Abb. 36 Schematische Darstellung des J2B-Clients.
Grüne Pfeile bedeuten Ausgaben für den Anwender, rote Pfeile stehen für Eingaben und deren Bearbeitung.

Dem Anwender stehen zwei Tools zur Kommunikation mit dem J2B-Client zur Verfügung. Beide Tools kommunizieren per RMI mit dem Dienst, trotzdem muss sich der Dienst auf demselben Rechner wie die Tools befinden. Dies ist keine Einschränkung von RMI, sondern soll sicherstellen, dass der Dienst auf dem lokalen Dateisystems des Auftraggebers alle nötigen Rechte besitzt, Dateien abzurufen und zu speichern, da der J2B-Client auch für Dateitransfers zwischen dem Anwenderrechner und dem ausführenden Rechner verantwortlich ist.

Das erste Tool ist ***J2B-Submit*** (Abb. 36, Schritt 1a). Es wird zusammen mit einem Parameter aufgerufen, der den Pfad zu einer Jobbeschreibung enthält. In das Verzeichnis, in dem diese Datei liegt, werden nach der erfolgreichen Jobausführung alle Ergebnisdateien kopiert. D.h. der Anwender muss in diesem Ver-

zeichnis Schreibrechte besitzen. J2B-Submit liest die Datei ein und schickt sie per RMI an ein Objekt von ServicesImpl des J2B-Client. Dieses veranlasst einen neuen Eintrag in der Joblist und schickt die generierte JobID an das aufrufende Programm zurück (2).

Diese JobID kann der Anwender zusammen mit dem zweiten Tool, ***J2B-State***, benutzen (1b). Er erhält dann jederzeit aktuelle Informationen über diesen Job. Im Falle eines Fehlers ist der Zustand **ERROR**, dann muss der Anwender den Job ggf. korrigieren und erneut abschicken. Im Erfolgsfall erhält der Anwender Auskunft darüber, in welchem Verzeichnis sich die Ergebnisdateien befinden.

Neben der Erstellung eines Joblist-Eintrags wird die Jobbeschreibung an den Provider weitergeleitet (3). Dieser erstellt aus der Jobbeschreibung eine oder mehrere JobClassAds. Beispiele für Job- und andere ClassAds finden sich im Anhang, Kapitel 13. Größtenteils werden die Bezeichner-Wert-Paare aus der Beschreibung 1:1 als Attribute in die JobClassAd übernommen. Bezeichner, für die das nicht gilt, werden weiter unten gesondert beschrieben. Zusätzlich werden noch Attribute mit dem Namen des Rechners (**NAME**), dessen RMI-Port (**PORT**) sowie dem Typ der ClassAd (**TYPE="JOB"**) eingefügt.

Besonders verfahren wird mit Attributen, die einen Dateitransfer vor dem eigentlichen Job nötig machen. Dazu gehören die Bezeichner der Jobbeschreibung **EXECUTABLE**, **IN** sowie **TRANSFER_INPUT_FILES**. Im Attribut **DATAFILES** der JobClassAd wird dazu eine Aufzählung aller nötigen Dateitransfers erstellt, da die Suche nach Datenquellen für diese Daten unabhängig von ihrer späteren Funktion bei der Ausführung des Jobs ist. Zunächst wird geprüft, ob eine benötigte Datei in dem Verzeichnis liegt, in dem auch die Jobbeschreibung gespeichert wurde. Ist die Datei vorhanden, so wird diese auch für den Job benutzt. Ansonsten wird während des Matchmakings ein passender Datenspeicher gesucht. Ein **DATAFILES**-Element hat folgende Attribute:

LABEL	Der Name des Elements, beginnend bei 'DATAFILE0' bis 'DATAFILEn'.
KIND	Gibt die Art des Elements an. Es wird unterschieden zwischen 'EXECUTABLE', 'IN' und 'DATA'.
FILE	Zusätzliche Informationen über die Datei. Standardmäßig hat **FILE** die Funktion **getFile** als Wert. Diese Funktion wird während des Matchens aufgerufen und schreibt zusätzliche Informationen über die Datei in die ClassAd. Siehe dazu auch Abschnitt 9.2.4.
REQUIREMENTS	Enthält die Bedingungen, die bei einem Match erfüllt ein müssen. Standardmäßig gilt **REQUIREMENTS=DATAFILEn.TYPE=="DATA" && FILE.isAvailable**, d.h. die matchende ClassAd muss vom Typ 'DATA' sein und die Datei muss dem DMS bekannt sein.

Tab. 8 Attribute eines Datafile-Elements einer JobClassAd

Die Bezeichner **REQUIREMENTS** und **RANK** aus der Jobbeschreibung werden in eine Unter-ClassAd **MACHINE** eingefügt, da diese die Bedingungen enthalten, die die matchende Machine erfüllen muss.

Zusätzlich gibt es noch globale **REQUIREMENTS**- und **RANK**-Attribute. Dieser Resourcebroker matcht nicht mehr nur eine Job- mit einer MachineClassAd sondern eine JobClassAd mit einer MachineClassAd und einer oder mehreren DataClassAds. Daher enthält die JobClassAd wie oben beschrieben einige Unter-ClassAds – nämlich den **MACHINE**-Teil sowie die einzelnen Elemente der **DATAFILES**-Aufzählung. All diese enthalten ein **REQUIREMENTS**-Attribut, welches beim Match mit einer zugehörigen ClassAd den Wahrheitswert „wahr“ ergeben muss.

Ziel dieser Arbeit ist ein Resourcebroker, der nicht nur die Jobausführung optimiert, sondern dabei auch die Datenverteilung mit in Betracht zieht. Die erwähnten **REQUIREMENTS**-Attribute sorgen nur dafür, dass die einzelnen Teile jeweils optimal sind. Würde keine weitere Optimierung stattfinden, hätte man zwar eine Ressource, die beispielsweise sehr leistungsfähig ist und Datenspeicher, die die benötigten Dateien besitzen. Trotzdem könnte eine Kombination „schlechterer“ Elemente besser sein, weil diese früher verfügbar sind oder eine höhere Datentransfergeschwindigkeit erlauben und so die Gesamtzeit des Jobs reduziert werden könnte. Daher gibt es in JobClassAds die globalen Attribute. Deren Werte kann der Benutzer nicht direkt festlegen sondern nur über die Bezeichner **DEADLINE**, **MAXCOSTS**, **MINCOSTS** und **MINTIME**. Sofern das **REQUIREMENTS**-Attribut schon vorhanden ist, wird die Ergänzung durch '**&&**' angehangen, da es eine zusätzliche Bedingung ist. Ist das **RANK**-Attribut schon vorhanden, so wird dieses mit dem zusätzlichen Wert multipliziert.

DEADLINE und **MAXCOSTS** sind Anforderungen, die eingehalten werden müssen und fließen daher in das **REQUIREMENTS**-Attribut ein. Die beiden anderen besagen, dass entweder Kosten oder die Ausführungszeit minimiert werden sollen und werden daher in das **RANK**-Attribut integriert. Die Erweiterungen an den beiden Attributen finden sich im Detail in der folgenden Tabelle.

DEADLINE	Gibt in der Jobbeschreibung einen Zeitpunkt an, zu dem der Job garantiert beendet sein muss. **REQUIREMENTS** wird ergänzt um '**MACHINE.Endtime <= DEADLINE**'.
MINTIME =true	Legt fest, dass die Gesamtausführungszeit von Beginn des ersten Datentransfers bis zum Ende des Jobs minimiert werden soll, daher wird **RANK** erweitert um '**1/(MACHINE.Endtime - minimum(DATAFILES[0].Starttime, ..., DATAFILES[n].Starttime))**'.
MAXCOSTS	Beschränkt die Maximalkosten. **REQUIREMENTS** wird erweitert um **COSTS <= MAXCOSTS**' wobei gilt: **COSTS=MACHINE.Costs+DATAFILES[0].Costs+...+DATAFILES[n].Costs**
MINCOSTS =true	Verlangt, dass die Gesamtkosten minimiert werden, d.h. **RANK** wird erweitert um '**1/COSTS**' wobei **COSTS** wie bei **MAXCOSTS** definiert ist.

Tab. 9 Erweiterungen der Attribute für Optimierungen

Nachdem die JobClassAd erstellt wurde, wird diese per RMI an die Services des Matchmakers geschickt (4). Wie erwähnt wird für jeden **QUEUE**-Befehl eine eigene ClassAd erstellt. Der J2B-Client erhält vom Matchmaker eine ClassAdID zurück, welche in der Joblist bei der zugehörigen JobID vermerkt wird.

Konnte eine JobClassAd gematcht werden, nutzt der Matchmaker eine RMI-Funktion der J2B-Client-Services um den Client darüber zu informieren. Die Services leiten diese Anfrage weiter an den Provider (5, 6). Der Provider ist dafür verantwortlich, die Jobausführung zu starten. Dazu erstellt dieser einen neuen Thread der Klasse ***Manager*** und übergibt dieser die Kontrolle über den Job. Der Manager schickt daraufhin per RMI den Befehl zur Jobausführung an die Services des M2B-Clients auf dem Rechner, auf dem der Job ausgeführt werden soll (7). Die Services des J2B-Clients bieten Funktionen, die es erlauben, Dateien vom aufgebenden zum ausführenden Rechner und zurück zu kopieren (8). Damit werden vor dem Job alle Eingabedateien, die lokal vorliegen, zum ausführenden Rechner kopiert sowie nach dem Job alle Ergebnisdateien und das Logfile zurückgesendet.

9.2.4 Benutzerdefinierte Funktionen

Das Condor ClassAd-API bietet eine Fülle eingebauter Funktionen. Darüber hinaus ist es möglich, eigene Funktionen zu implementieren. Dazu sind zwei Schritte erforderlich: Zunächst müssen die eigenen Funktionen in einer beliebigen Klasse implementiert werden. Die Funktionen müssen **public** und **static** sein. Parameter und Rückgabewert müssen vom Typ **Expr**[48] sein, der Parameter kann auch **Expr[]** sein.

Als zweiter Schritt muss die Klasse **condor.classad.FuncCall** um den folgenden Code erweitert werden:

```
static {
  loadJavaLibrary("eigeneklasse")
}
```

wobei ***'eigeneklasse'*** der Name der Klasse ist, die die benutzerdefinierten Funktionen enthält. Anschließend stehen die Funktionen dem ClassAd-API zur Verfügung und können in ClassAds benutzt werden.

Im Rahmen dieser Arbeit wurden die folgenden benutzerdefinierte Funktionen erstellt, zu finden in der Datei **broker.MM.Builtin.java**:

- **Expr minvalue(Expr[])**

[48] **Expr** bezeichnet im ClassAd-API einen beliebigen Ausdruck, siehe auch Anhang, Abschnitt 13.6.

- **Expr maxvalue(Expr[])**
- **Expr getFile(Expr dataClassad, Expr filename, Expr starttime, Expr host)**
- **Expr getCosts(Expr host, Expr starttime, Expr duration)**
- **Expr checkMachine(Expr host, Expr minstart, Expr duration, Expr maxend)**

minvalue und **maxvalue** geben den Ausdruck aus der Liste der übergebenen Ausdrücke zurück, der am kleinsten bzw. am größten ist. Die Ausdrücke können dabei Ganzzahlen, Gleitkommazahlen oder Datumswerte sein, wobei nicht alle Ausdrücke vom gleichen Typ sein müssen, da die Funktion gegebenenfalls alle Werte in Gleitkommazahlen umwandelt. Sollte einer der Ausdrücke von einem anderen Typ sein, wird **ERROR** zurückgegeben, ist die Liste leer, so ist das Resultat **UNDEFINED**.

getFile prüft, ob das DMS, dass durch die DataClassAd **dataClassAd** beschrieben wird, eine Datei mit dem angegebenen Namen **filename** besitzt. Für die Abschätzung der Transferzeit und der Transferkosten kann eine Startzeit angegeben werden, zu der die Aktionen beginnen sollen sowie ein möglicher Zielrechner. Die Funktion dient dazu, während des Matchmakings Datenmanagementsysteme dynamisch abzufragen. **getFile** liefert immer eine ClassAd zurück, die mindestens das Attribut **isAvailable** enthält, welches angibt, ob die Datei vorhanden ist (**true**) oder nicht (**false**). Für den Fall, dass die Datei vorhanden ist, gibt es weitere Attribute: **nsl** und **transfertime** geben den tatsächlichen Speicherort und die geschätzte Transferzeit zur Ressource **host** an, **costs** die geschätzten Kosten für den Datentransfer.

getCosts erfragt beim M2B-Client der Ressource **host** die Kosten für eine Jobausführung in Abhängigkeit von der Startzeit **starttime** und der wahrscheinlichen Dauer. Zurückgegeben wird ein Zahlenwert, die die Kosten repräsentiert.

checkMachine prüft die Verfügbarkeit eines Rechners während des Matchmakings. Als Parameter werden die früheste mögliche Startzeit **minstart**, die Dauer des Jobs **duration** sowie die späteste erlaubte Endzeit **maxend** angegeben. Anschließend wird der M2B-Client **host** kontaktiert, welcher die Reservierungsanfragen und die geplanten Jobs prüft und die Start- und Endzeit (**starttime** bzw. **endtime**) des nächstmöglichen Beginns des Jobs in der vorgegebenen Zeitspanne zurückgibt. Ein Attribut **isAvailable** gibt an, ob die Ressource überhaupt zur Verfügung steht.

9.2.5 Der M2B-Client

Der M2B-Client hat die Aufgaben, eine Ressource im System bekanntzumachen und Jobaufträge anzunehmen und diese auszuführen. Daher werden beim Start eines M2B-Clients zwei Threads gestartet, einer vom Typ *Provider* und einer vom Typ *Scheduler*.

Der Provider erstellt auf Basis von gemessenen und ausgelesenen Daten eine MachineClassAd und schickt diese an den Matchmaker (Abb. 37, Schritt 1). In der aktuellen Implementation werden die Eigenschaften aus einer Datei eingelesen, die der Besitzer der Ressource erstellen und dem M2B-Client als Startparameter übergeben muss. Die Datei enthält eine Menge von Bezeichner-Wert-Paaren, die 1:1 in die ClassAd übernommen werden. Für die Zukunft ist es denkbar, dass die Eigenschaften durch verschiedene Module erfasst und dem Provider zur Verfügung gestellt werden. Der Provider startet einen Thread vom Typ *SendClassThread*, der periodisch aus den in einer Hashtable gespeicherten Attributen eine MachineClassAd erstellt und an den Matchmaker sendet (2).

Der Scheduler verwaltet in verschiedenen Listen alle wartenden, ausgeführten und abgebrochenen Jobs. Zusätzliche kann er temporäre Reservierungen für die Ressource annehmen (3). Wird eine gematchte ClassAd von einem J2B-Client an einen M2B-Client geschickt, so benachrichtigen die *Services* den Scheduler, welcher daraufhin zunächst ein *Jobcontrol*-Objekt erstellt (4).

Dieses repräsentiert den Job und ist für dessen Vorbereitung und Ausführung verantwortlich. Um zu verhindern, dass Fehler in einem Job den gesamten M2B-Client zum Absturz bringen, werden alle Jobcontrols in einem eigenen Thread gestartet. Das erstellte Objekt wird in die Liste der wartenden Jobs aufgenommen (5) und erhält den Auftrag, die Vorbereitungen für den Job zu starten (6). D.h. das Jobcontrol erstellt ein *FileTransfer*-Objekt für jede Datei, die übertragen werden muss, egal ob diese Datei lokal beim Auftraggeber oder bei einem DMS zu finden ist. Wurden alle Datentransfers durch die FileTransfer-Objekte abgeschlossen, wird ein Flag gesetzt, dass die Vorbereitungen beendet wurden. Eine Erweiterung des FileTransfer-Objekts ist denkbar, so dass dieses auch in der Lage ist, abgebrochene Übertragungen wieder aufzunehmen oder eine Datei von mehreren Quellen gleichzeitig zu lesen.

Der Scheduler prüft permanent die Liste der wartenden Jobs, welche nach den Anfangszeiten sortiert sind. Wurde die Startzeit des ersten wartenden Jobs erreicht und sind dessen Vorbereitungen erfolgreich abgeschlossen worden, wird dieser gestartet (7). Wird zu diesem Zeitpunkt noch ein Job ausgeführt, wird dieser abgebrochen und die Ausführungsumgebung bereinigt. In diesem Fall muss der Auftraggeber den Jobauftrag erneut an den Matchmaker schicken, eine automatische Neuplatzierung ist nicht implementiert. Auf einer Ressource kann immer nur genau ein Job ausgeführt werden, es ist jedoch möglich, dass sich mehr als ein Job in der Vorbereitungsphase befindet.

Erhält ein Jobcontrol den Befehl, den Job zu starten, wird eine Ausführungsumgebung erstellt und die Ein- und Ausgabeströme entsprechend den Angaben der MatchedClassAd umgelenkt. Anschließend wird das Programm gestartet und gewartet, bis dieses sich beendet. Weitere Überwachungsmöglichkeiten gibt es in der aktuellen Implementation nicht. Die Ausführung kann aber abgebrochen werden, wenn der Scheduler den Auftrag dazu gibt, etwa weil der Job nicht in der vorgesehenen Zeit beendet werden konnte. Im Falle einer erfolgreichen Ausführung beginnt das JobControl alle Dateien, die in der MatchedClassAd angegeben wurden, zurück zum Auftraggeber zu senden (8). Anschließend werden alle lokal erstellten Dateien gelöscht.

Scheduler und Jobcontrol kontaktieren regelmäßig den J2B-Client, von dem ein Job stammt, um den Zustand des Jobs und die damit verbundenen Informationen zu aktualisieren.

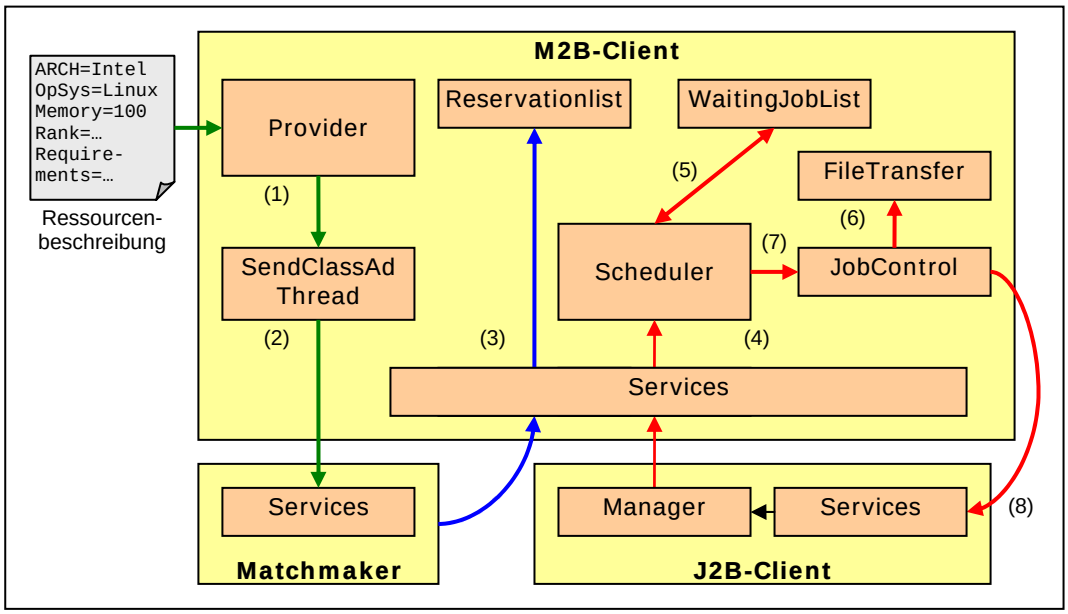


Abb. 37 Schematische Darstellung des M2B-Clients
Unterscheidung von drei Phasen: Bekanntmachung der Ressource (grüne Pfeile), Reservierung (blaue Pfeile) und Jobausführung (rote Pfeile).

9.2.6 Der D2B-Client

Der D2B-Client startet zwei Threads, *Provider* und *Adapter*. Provider erstellt periodisch eine ClassAd mit Informationen über das zugeordnete Datenmanagementsystem und schickt dieses an den Matchmaker. In der aktuellen Implementation werden die Informationen des DMS aus einer Datei eingelesen. Der Adapter ist die eigentliche Schnittstelle zwischen DMS und dem Resourcebroker, wodurch für jedes DMS eine eigene Implementation erstellt werden muss. Vom Resourcebroker nimmt der Adapter Anfragen über die RMI-Schnittstelle entgegen, wandelt ggf. die übergebenen Parameter um und formuliert eine CORBA-Anfrage an das ZIBDMS. Die benötigten CORBA-Klassen wurden automatisch aus der

IDL-Datei des ZIBDMS mit dem Programm **idlj** des Java Development Kits generiert. Erhält der Adapter Ergebnisse der CORBA-Anfrage zurück, werden diese in ein für den Resourcebroker kompatibles Format gewandelt und die RMI-Anfrage beantwortet.

9.2.7 Der Matchmaker

Als zentrale Instanz ist der Matchmaker dafür verantwortlich, ClassAds von Ressourcen, Datenspeichern und Jobs zu sammeln und unter diesen passende Kombinationen zu suchen. „Passend“ heißt hierbei, dass alle **REQUIREMENTS**-Attribute der beteiligten ClassAds zu dem Wahrheitswert „wahr“ ausgewertet werden können. Startet man den Matchmaker, so wird eine neue Instanz der Klasse *Broker*[49] erstellt. Diese erstellt sechs Listen zur Verwaltung des ClassAds. Je Typ gibt es eine Liste für neue ClassAds, die bei noch keinem Matchversuch beteiligt waren, und eine für solche, die schon ohne Erfolg gematcht wurden. Die Listen sind vom Typ *ClassAdList* und speichern eine Menge von *ClassAdListItems* in einem *Vector*. Darüber hinaus bieten die ClassAdLists noch spezielle Funktionen zur Manipulation von ClassAdListItems an, beispielsweise gibt es Funktionen, die die Gültigkeit von ClassAds überprüfen.

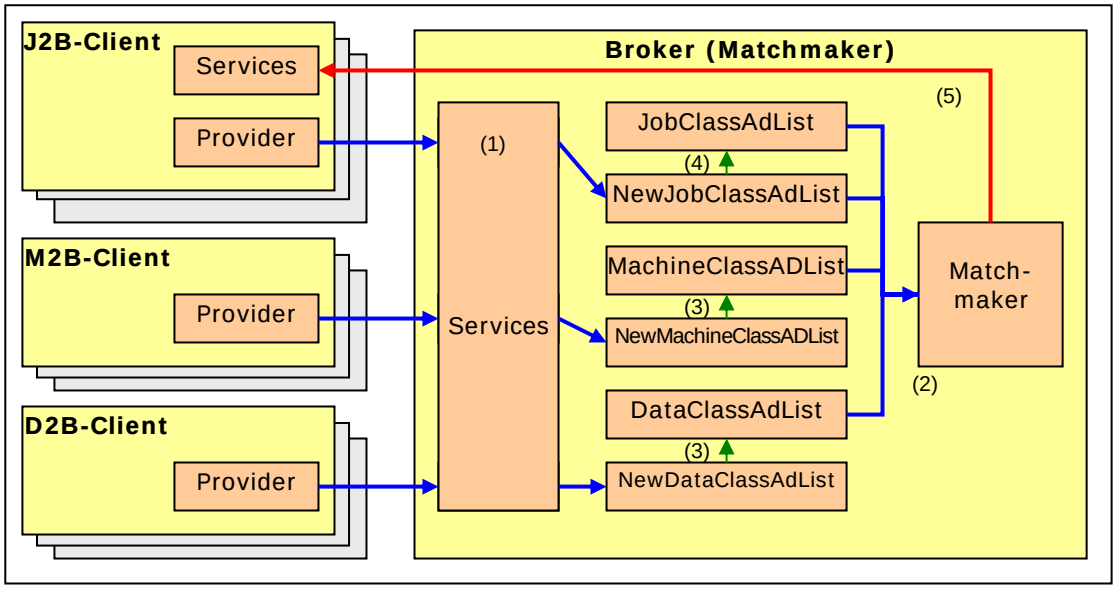


Abb. 38 Schematische Darstellung des Matchmakers
Blaue Pfeile zeigen den Weg der ClassAds an. Grüne Pfeile zeigen mögliche Wechsel von ClassAds zwischen einzelnen Listen an. Der rote Pfeil stellt die Benachrichtigung des J2B-Clients bei einem Match dar.

Es wird dann ein Objekt der Klasse *Matchmaker* erstellt und gestartet, welche für das eigentliche Matchen verantwortlich ist. Das Objekt wird benachrichtigt,

[49] Der Name „Matchmaker“ wird für eine Teilkomponente benutzt, die nur für das eigentliche Matchen verantwortlich ist, daher wird die Klasse des Startobjekts als „Broker“ bezeichnet.

sobald eine der Listen ein neues Element erhält bzw. ein Element aktualisiert wird (Abb. 38, Schritt 2). Die *Services*-Klasse, welche RMI-Anfragen bearbeitet, bietet fünf Funktionen an, die der Übermittlung bzw. der Aktualisierung der verschiedenen ClassAds dienen. Zunächst wird ein ClassAdListItem erstellt und der jeweiligen Liste hinzugefügt (1).

In Abschnitt 8.3.4 wurde aufgezählt, wann der Matchmaker einen neuen Matchversuch unternimmt. Implementiert wurde dies durch die Trennung der neuen ClassAds von den anderen. Der Matchmaker versucht nur einen Match, wenn es neue bzw. aktualisierte ClassAds gibt. Je nach ClassAd-Typ unterscheidet sich jedoch, welche ClassAds miteinander gematcht werden. Gibt es eine neue Machine- oder DataClassAd, so wird diese einfach in die Liste der schon bekannten ClassAds einsortiert (3) und es werden alle JobClassAds durchlaufen, um diese zu matchen. Dabei werden alle bekannten Machine- und DataClassAds getestet. Gibt es eine oder mehrere neue JobClassAds, wird versucht, diese mit den Machine- und DataClassAds zu matchen. Die schon bekannten JobClassAds werden nicht betrachtet, da diese bisher nicht gematcht werden konnten und sich die Möglichkeiten für diese nicht geändert haben. Konnte eine solche neue JobClassAd nicht gemacht werden, wird sie in die Liste der bekannten JobClassAds überführt (4), um die Auslastung der Ressource, auf der der Matchmaker ausgeführt wird, zu reduzieren.

Das Matchmaking funktioniert, wie es der Algorithmus in der Abb. 29 beschreibt. In einer Liste werden zunächst alle möglichen MatchedClassAd gesammelt und nach ihren Rangwerten sortiert. Diejenige an der ersten Stelle wird dann für die Jobausführung ausgewählt und der zugehörige J2B-Client wird per RMI über den Match informiert. Die JobClassAd wird dann aus den Listen des Matchmakers gelöscht und es wird nach weiteren möglichen Matches gesucht (5).

Zu erwähnen ist noch die Klasse *Builtin*, die die Implementationen der in Abschnitt 9.2.4 vorgestellten Funktionen enthält.

10 Leistungsmessungen

Im Folgenden werden die Ergebnisse einiger Tests des neuen Resourcebrokers präsentiert. Es soll vor allem untersucht werden, ob der zusätzliche Aufwand des erweiterten Matchmakings, insbesondere die Abfrage externer Systeme, im Verhältnis zu den erreichten Zeit- bzw. Kosteneinsparungen steht. Da in der aktuellen Implementation die Ausführungszeit des eigentlichen Jobs fest in der JobClassAd vorgegeben ist, konzentriert sich dieses Kapitel auf die Auswirkungen bei der Auswahl des besten Replikats.

10.1 Aufbau der Testumgebung

Um eine wirklichkeitsnahe Umgebung mit realistischen Antwortzeiten zu simulieren, sollten die einzelnen Komponenten möglichst auf Rechnern ausgeführt werden, die sich nicht im gleichen LAN befinden. In verschiedenen Messungen wurde eine durchschnittliche Antwortzeit von 40 ms je WAN-Verbindung ermittelt, was einen erhöhten Zeitbedarf beim Matchmaking, bedingt durch die Abfrage externer Systeme, nach sich zieht.

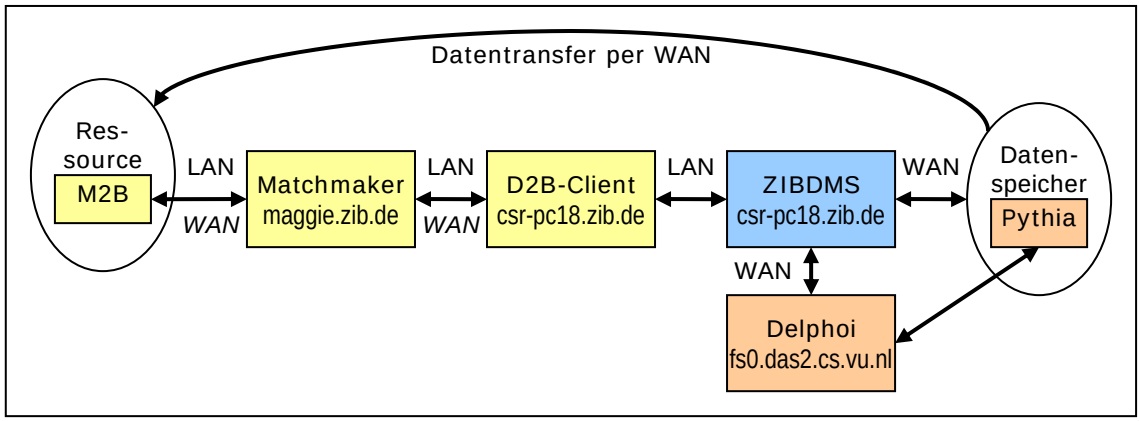


Abb. 39 Aufbau der Testumgebung
Die Beschriftungen an den Pfeilen geben an, ob sich bei der Testumgebung die Rechner im gleichen LAN befanden oder nicht. Die kursiven Beschriftungen unter den Pfeilen zeigen abweichende Verbindungswege in der Realität an.

Abb. 39 zeigt den Aufbau der Testumgebung. Der Resourcebroker und das ZIBDMS befinden sich hierbei im gleichen LAN, Delphoi und die benutzen Pythias sind über WAN-Verbindungen erreichbar. Benötigt der Matchmaker die beste Quelle einer Datei, so schickt dieser eine Anfrage an den verantwortlichen D2B-Client, welcher die Anfrage an das entsprechende ZIBDMS weiterleitet. Dort wird eine Liste aller Replikate zusammengestellt und Delphoi zur Schätzung der Übertragungszeiten übergeben. Als Delphoi-Server fungiert der frei verfügbare Rechner fs0.das2.cs.vu.nl, welcher Pythias in verschiedenen WANs verwaltet. Die für die Tests benutzten Pythias finden sich in Tab. 10. Da keine Schreibrechte auf diesen

Rechnern vorlagen, konnten keine Dateien dort angelegt und somit auch keine tatsächlichen Datentransfers durchgeführt werden. Im ZIBDMS können aber Replikate auf diesen Rechnern registriert werden, ohne dass diese dort tatsächlich existieren müssen. Somit kann in der Testumgebung das Matchmaking durchgeführt werden, eine Ausführung der Jobs ist in dieser jedoch nicht möglich.

Rechnername	Kurzform
ia64.icis.pcz.pl	ia64
pegasos.icis.pcz.pl	pegasos
n0.hpcc.sztaki.hu	n0
mike4.cct.lsu.edu	mike4
fs0.das2.cs.vu.nl	fs0
fs1.das2.liacs.nl	fs1
rage1.man.poznan.pl	rage1
skirit.ics.muni.cz	skirit
eltoro.pcz.pl	eltoro
csr-pc18.zib.de	csr-pc18

Tab. 10 Namen der Rechner, die als Datenspeicher fungierten

10.2 Test: Skalierbarkeit von Delphoi

Testziel: Mit diesem Test soll untersucht werden, wie sich die Antwortzeit einer Delphoi-Anfrage in Abhängigkeit der Anzahl der Pythias, die zur Beantwortung nötig sind, verhält.

Während des Matchmakings wird an die Netzwerkinformationsdienste der einzelnen Datenmanagementsysteme eine Vielzahl von Anfragen gestellt. Um die Zahl dieser Anfragen zu reduzieren, ist es erforderlich, dem Dienst eine Liste aller bekannten Replikate zu übergeben, für die dieser dann die notwendigen Berechnungen ausführt. Delphoi wendet sich dazu an die einzelnen Pythias um die benötigten Daten zu erfragen. Diese Anfragen sollte asynchron geschehen, um parallel möglichst viele Pythias abfragen zu können.

Testablauf: Für den Test wurde im ZIBDMS eine Datei angelegt und ihr eine einzige NSL zugeordnet. Anschließend wurde eine Jobbeschreibung erstellt, die genau diese als einzige Datei zur Ausführung benötigt und dem Matchmaker übergeben. Dieser versucht während des Matchmakings über den D2B-Client die beste Speicherquelle für diese Datei zu erhalten. Für den Fall, dass nur eine Speicherquelle zur Verfügung steht, muss das DMS keine Anfrage an Delphoi stellen. Für diesen Test wurde das ZIBDMS jedoch so angepasst, dass es auch bei nur einem verfügbaren Replikat eine Delphoi-Anfrage stellt, da hier die Antwortzeit in Abhängigkeit der Anzahl der Pythias interessant ist. Anschließend wurde die Zahl der Replikate auf zwei, vier und schließlich acht erweitert.

Testergebnis: Abb. 40 zeigt die Ergebnisse der Messung. Für die Messungen mit einem Replikat wurden zunächst mike4 und anschließend n0 benutzt. Die Anfragen benötigten im Mittel 1802 bzw. 1314 ms. Für die Messung mit zwei Replikaten wurden ebenfalls diese beiden Ressourcen benutzt. Die Anfragezeit erhöhte sich hierbei im Durchschnitt auf 2817 ms, was nur geringfügig unter der Summe der beiden Einzelergebnisse liegt. Auch bei vier bzw. acht Replikaten ist die Anfragzeit weiter linear angestiegen, wie es die hellblaue Gerade zeigt, die aus dem Mittelwert aller Anfragen berechnet wurde.

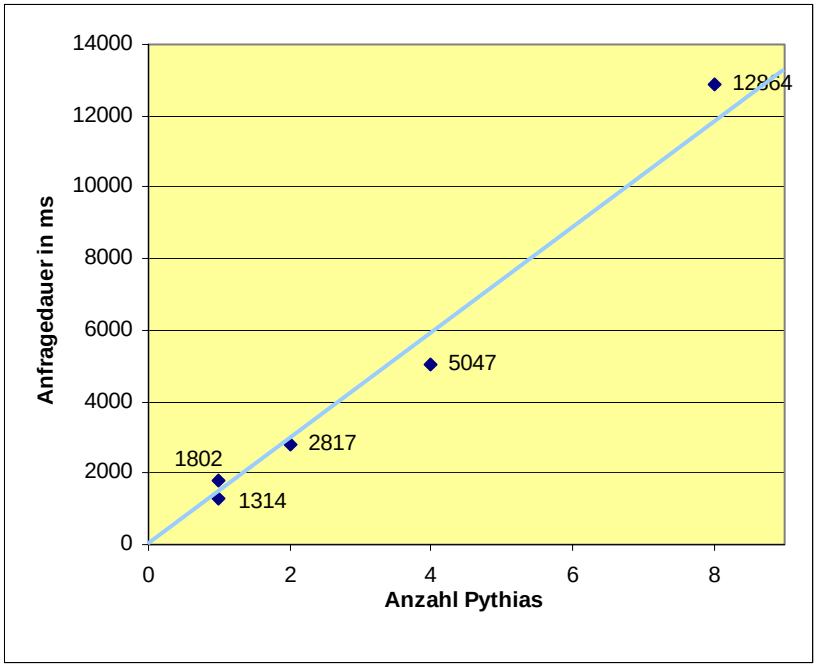


Abb. 40 Dauer einer Delphoiabfrage in Abhängigkeit der Anzahl der Pythias

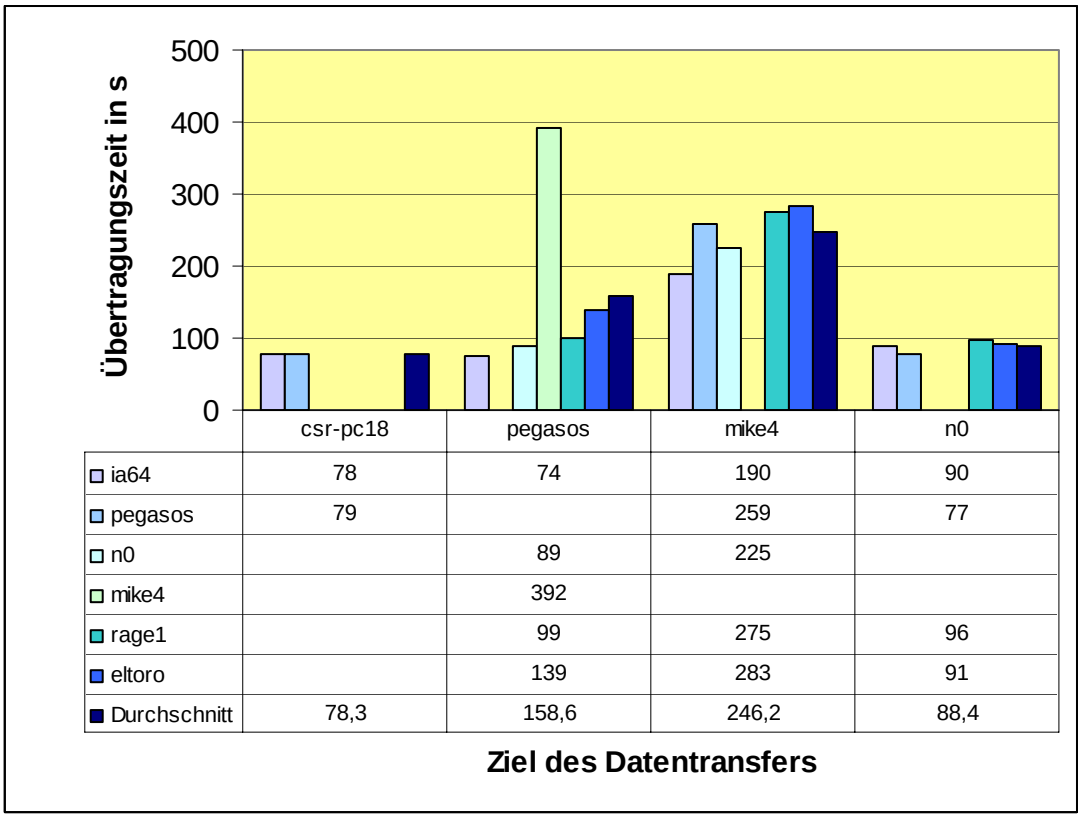
Resultat: Es zeigt sich also, dass die Anfragezeit bei Delphoi linear zur Anzahl der beteiligten Pythias ansteigt. Da davon ausgegangen werden kann, dass je Job mehr als eine Datei benötigt wird und für jede dieser Dateien mehrere Replikate vorhanden sind, sinkt die Geschwindigkeit beim Matchmaking gegenüber der ursprünglichen Condor-Version. Im folgenden Test wird daher untersucht, wie groß die Zeitersparnis beim Datentransfer durch die Nutzung von Delphoi ist. Um die Anfragezeiten zu reduzieren ist es auch möglich, die Ergebnisse der Anfragen lokale beim D2B-Client oder direkt beim Matchmaker zwischenzuspeichern.

10.3 Test: Berechnung der Ausführungszeit

Testziel: Im vorigen Abschnitt wurde gezeigt, dass die Zeiten zur Abfrage von Delphoi nicht unerheblich sind. Daher soll in diesem Abschnitt untersucht werden, inwieweit sich die Übertragungszeiten von Replikaten, die mit Delphoi ausgewählt wurden, von Replikaten, die zufällig ausgewählt wurden, unterscheiden.

Testablauf: Dazu wurde eine Jobbeschreibung erstellt, die eine Datei mit einer Größe von einem Gigabyte benötigt. Diese Datei wurde im ZIBDMS eingetragen und acht Datenspeicher (ia64, pegasos, n0, mike4, rage1, eltoro, fs0, fs1 und skirit) zugeordnet. Es war für jeden Test immer nur genau eine ausführende Ressource (csr-pc18, pegasos, mike4 bzw. n0) verfügbar, zu der die Datei übertragen werden sollte. Während des Matchmakings wurden Anfragen an Delphoi gestellt, um das beste Replikat auszuwählen. Zusätzlich wurden die geschätzten Übertragungszeiten der anderen Replikate mitprotokolliert.

Testergebnis: In allen Fällen wurde immer das schnellste Replikat für die Übertragung ausgewählt. Drei Datenspeicher (fs0, fs1 und skirit) lieferten jedoch für keine Anfrage ein verwertbares Ergebnis zurück. Vermutlich wurde das Modul PathChirp nicht ausgeführt oder es standen keine Ergebnisse auf Grund von Fehlern bei der Messung zu Verfügung. Nur zwei der restlichen sechs Datenspeicher (ia64 und pegasos) konnten Angaben zu allen vier möglichen Zielen machen.



	csr-pc18	pegasos	mike4	n0
ia64	78	74	190	90
pegasos	79		259	77
n0		89	225	
mike4		392		
rage1		99	275	96
eltoro		139	283	91
Durchschnitt	78,3	158,6	246,2	88,4

Abb. 41 Von Delphoi geschätzte Übertragungszeiten
Die Quellen der Datentransfers sind links angegeben, die Ziele unter dem Diagramm.

Es ist ersichtlich, dass die Übertragungszeiten verschieden stark variieren. Für csr-pc18 lohnt sich in diesem Fall die Delphoi-Anfrage nicht, es stehen jedoch nur sehr wenige Quellen zur Verfügung. Die geschätzten Zeiten für Datentransfers zu

pegasos variieren stark. Selbst die Wahl eines zufälligen Replikats empfiehlt sich nicht, da der Transfer von ia64 mit 74 Sekunden wesentlich schneller ist als die durchschnittliche Transferzeit aller Datenquellen von 159 Sekunden. Bei den beiden anderen Ressourcen zeigen sich ähnliche Ergebnisse.

Resultat: Der Test zeigt, dass sich der zusätzliche Aufwand durch die Delphoi-Anfragen ab einer bestimmten Datenmenge lohnt. Es ist zu beachten, dass die gewählte Dateigröße von einem Gigabyte für datenintensive Jobs im Grid als sehr gering eingestuft werden kann. In den meisten Fällen hat sich aber schon hier der Einsatz von Delphoi gelohnt.

10.4 Test: Replikatauswahl bei mehreren DMSen

Testziel: Der Resourcebroker dieser Arbeit ist in der Lage, mit mehr als einem Datenspeicher zusammenzuarbeiten und kann die Daten für einen Job von verschiedenen Datenspeichern beziehen. Hierbei unterscheidet sich der Broker vor allem vom Gridbus Broker, der in Abschnitt 2.2 vorgestellt wurde, da dieser alle Daten auf einem Datenspeicher erwartet. Daher wurde auf Vergleichstest mit diesem Broker verzichtet.

In diesem Test soll untersucht werden, ob der Resourcebroker tatsächlich das Replikat mit der geringsten Übertragungszeit auswählt, unabhängig davon, über welches Datenmanagementsystem diese Datei erreichbar ist.

Testablauf: Beim Matchmaker werden zwei verschiedene Datenmanagementsysteme registriert. Beiden sind Replikate einer bestimmten Datei bekannt, die für den auszuführenden Job benötigt werden. Die Replikate wurden auf Basis der Ergebnisse von Test 10.3 registriert. Ziel eines möglichen Datentransfers ist der Rechner pegasos. Das erste ZIBDMS kennt ein Replikat der ersten Datei auf ia64 (Transferzeit: 74s) und eines der zweiten Datei auf eltoro (139s). Dem zweiten ZIBDMS ist für die erste Datei ein Replikat auf mike4 (392s) und für die zweite Datei auf n0 (89s) bekannt.

Testergebnisse: Der Matchmaker wählt für die erste Datei das Replikat auf ia64 aus, welches über das erste ZIBDMS ermittelt wurde. Für die zweite Datei wird das Replikat von n0 benutzt, da dieses gegenüber mike4 eine kürzere Übertragungszeit bietet.

Resultat: Wie erwartet wählt der Matchmaker das Replikat einer Datei mit der geringsten Übertragungszeit aus, egal über welches DMS das Replikat gefunden wurde. Zudem können innerhalb eines Matches Replikate

11 Zusammenfassung

Die dieser Arbeit vorangegangene Studienarbeit hat sechs verschiedene Resourcebroker miteinander verglichen und dahingehend untersucht, inwieweit diese die Verteilung der für einen Job benötigten Daten mit in die Ressourceauswahl einbeziehen. Der in dieser Hinsicht vielversprechenste Broker, der Gridbus Broker, wurde zu Beginn dieser Arbeit nochmals vorgestellt. Er ist in der Lage, ein Paar aus Datenspeicher und ausführender Ressource so zu wählen, dass die Summe aus Ausführungs- und Datentransferzeit minimiert wird.

Beim Gridbus Broker können für einen Job aber immer nur die Daten eines einzigen Datenspeichers benutzt werden. In der Praxis hingegen werden häufig Daten von verschiedenen Datenspeichern für einen Job benötigt. So ist im angesprochenen C3-Grid die Nutzung von Daten anderer Forschungseinrichtungen ein ausdrücklich erwähntes Ziel.

Im Rahmen dieser Arbeit wurde daher ein Resourcebroker entwickelt, der für einen Job Daten von verschiedenen Datenspeichern beziehen kann. Der Resourcebroker ist neben der Auswahl der entsprechenden Ressourcen in der Lage, die Ausführungszeit bzw. die für die Ausführung anfallenden Kosten zu minimieren.

Im theoretischen Teil wurde dazu nach der Definition wichtiger Begriffe ein formales Modell für den Resourcebroker entworfen. Es wurden die einzelnen Komponenten spezifiziert sowie die Optimierungsprobleme für die Gesamtzeit und die Gesamtkosten aufgestellt.

Als Vorbereitung auf den praktischen Teil wurden im weiteren Verlauf verschiedene Systeme vorgestellt, die dem Resourcebroker als Basis dienen bzw. die Konzepte präsentierten, die im Broker umgesetzt wurden. Es wurde ausführlich die ClassAd-Sprache vorgestellt, die zur Beschreibung von Ressourcen im Grid dient. Der Resourcebroker benutzt die Sprache zur Beschreibung von Jobaufträgen, Ressourcen und Datenspeichern. Des Weiteren wurde das Suchen und Zusammenfügen passender ClassAds – das Matchmaking – beschrieben. Die im formalen Modell spezifizierten Optimierungen können in die Anforderungen der einzelnen ClassAds integriert werden, wodurch der Matchmaker die optimale Kombination aus ClassAds auswählen kann.

Weiterhin wurde beschrieben, wie benutzerdefinierte Funktionen in das ClassAd-API integriert werden können, wodurch es möglich wird, während des Matchmakings externe Systeme zur Gewinnung dynamischer Informationen abzufragen. Der Matchmaker des Resourcebroker ist dadurch in der Lage, die aktuelle Auslastung der einzelnen Ressourcen zu ermitteln und Informationen darüber einzuholen, zu welchem Zeitpunkt eine Ressource wieder verfügbar ist. Zum anderen kann der Matchmaker über diese Funktionen die Speicherorte und die geschätzten Transferzeiten für benötigte Dateien abfragen.

Diese Abfragen werden an ein Datenmanagementsystem weitergeleitet, welches für die Verwaltung von Replikaten verantwortlich ist. In dieser Arbeit wurden zunächst Datenmanagementsysteme allgemein vorgestellt, um anschließend eine Realisierung – das ZIBDMS – zu präsentieren. Im Rahmen dieser Arbeit wurde das ZIBDMS um eine Funktion erweitert, die für eine Datei das Replikat ermittelt, dessen geschätzte Übertragungszeit zu einem bestimmten Ziel am geringsten ist. Die nötigen Informationen dafür erhält es von einem Netzwerkinformationsdienst, in diesem Fall von Delphoi. Er kontrolliert eine Menge von Pythias, d.h. von Programmen die auf den einzelnen Ressourcen ausgeführt werden und Informationen über diese und deren Netzwerkverbindungen sammeln. Delphoi kann aus den gemessenen Daten Vorhersagen über die zukünftige Entwicklung der einzelnen Eigenschaften treffen und anderen Anwendungen wie beispielsweise dem ZIBDMS zur Verfügung stellen. Neben diesen drei Projekten, die bei der Implementation benutzt wurden, sind in dieser Arbeit noch drei weitere Projekte auf der Basis von ClassAds vorgestellt wurden, welche teilweise ähnliche Ansätze wie der Broker dieser Arbeit verfolgen, sich aber nicht für eine Erweiterung geeignet haben. Die Gründe dafür wurden in der Arbeit aufgezeigt.

Im praktischen Teil wurde der Resourcebroker in Java implementiert. Wie bei Grid-Systemen typisch, wurde dieser modular aufgebaut. Als zentrale Instanz fungiert der Matchmaker, welcher Informationen in Form von ClassAds von den einzelnen Clients erhält. Die Clients dienen als Schnittstelle zwischen dem Anwender, der Ressource bzw. dem Datenspeicher auf der einen Seite und dem Matchmaker auf der anderen Seite.

Die wichtigsten Aspekte des neuen Resourcebrokers sind die Einbeziehung von Informationen über die Verteilung von Daten, welche über Datenmanagementsysteme erfragt werden können und die Einführung von globalen Attributen in ClassAds. Diese ermöglichen es, Optimierungen für alle Teile einer ClassAd festzulegen, wodurch Ressourcen und Datenspeicher für einen Job so gewählt werden können, dass dessen Gesamtausführungszeit bzw. dessen Gesamtkosten minimiert werden.

Die Leistungsmessungen haben gezeigt, dass sich die Einbeziehung der Datenverteilung in die Jobplanung positiv auf die Gesamtausführungszeit eines Jobs auswirkt. Anfragen an den Netzwerkinformationsdienst Delphoi benötigen jedoch viel Zeit. Zudem skaliert Delphoi bei zunehmender Anzahl der abzufragen Pythias schlecht. Abhilfe schaffen würde hier eine asynchrone Abfrage der Pythias durch Delphoi oder eine Zwischenspeicherung der Ergebnisse beim Matchmaker bzw. bei den entsprechenden Schnittstellen. Bei typischen Dateigrößen im Grid von mehreren Gigabyte fallen die Anfragezeiten im Verhältnis zur Zeitersparnis durch die kürzeren Datentransfers nicht ins Gewicht. Das heißt, dass die Einbeziehung der Datenverteilung in die Jobplanung lohnenswert ist.

12 Literaturverzeichnis

[1] D. Abramson, R. Buyya, & J. Giddy. *A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker*. Future Generation Computer Systems (FGCS) Journal, 18(8), 1061-1074, Elsevier Science, The Netherlands, 2002.

[2] D. Abramson, J. Giddy & F. Kotler. *High Performance Parametric Modeling with Nimrod/G: Killer Application fort he Global Grid?* Proceedings of the International Parallel and Distributed Processing Sympsosium IPDPS 2000, IEEE CS Press, USA, 2000.

[3] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, & I. Taylor. *Enabling Applications on the Grid – A GridLab Overview*. International Journal of High Performance Computing Applications, 17(4), 449-466, 2003.

[4] G. Aloisio, M. Cafaro, E. Blasi, & I. Epicoco. *The Grid Resource Broker, a Ubiquitous Grid Computing Framework*. Journal of Scientific Programming, Special Issue on Grid Computing, IOS Press, Amsterdam, The Netherlands.

[5] G. Aloisio, M. Cafaro, P. Falabella, C. Kesselman, & R.Williams. *Grid Computing on the Web Using the Globus Toolkit.* In Proceedings of the HPCN Europe 2000, Amsterdam, The Netherlands, Lecture Notes in Computer Science, Springer, 1823, 32-40, 2000.

[6] A. Baranovski, G. Garzoglio, I. Terekhov, A. Roy, & T. Tannenbaum. *Management of Grid Jobs and Data within SAMGrid*. The 2004 IEEE International Conference on Cluster Computing, San Diego, CA, USA. 323-360. 2004.

[7] R. Buyya, D. Abramson, J. Giddy & Heinz Stockinger. *Economic Models for Resource Management and Scheduling in Grid Computing*. Special Issue on Grid Computing Environments, The Journal of Concurrency and Computation: Practice and Experience (CCPE), 14, 13-15, 1507-1542, Wiley Press, USA, 2002.

[8] C3-Grid Projektantrag.

[9] S. Cavalieri, & S. Monforte. *Resource Broker Architecture and APIs*. 2001. http://server11.infn.it/workload-grid/docs/20010613-RBArch-2.pdf.

[10] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury & S. Tuecke. *The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets.* Journal of Network and Computer Applications, 23:187-200, 2001.

[11] The ClassAd Language Reference Manual. http://www.cs.wisc.edu/condor/classad/refman.pdf.

[12] Condor ClassAd-API, Javadoc. http://www.cs.wisc.edu/condor/classad/javadoc/V2.2.

[13] *Condor Version 6.7.7 Manual*. Condor Team, University of Wisconsin-Madison, WI, USA, 2005.

[14] D. Davis & M. Parashar. *Latency Performance of SOAP Implementations*. Proceedings of the CCGRID'02, 2002.

[15] C. Dovrolis, P. Ramanathan & D. Moore. *What do Packet Dispersion Techniques Measure?* In IEEE Infocom, 2001.

[16] D. Dullmann, W. Hoschek, J. Jean-Martinez, A. Samar, H. Stockinger & K. Stockinger. *Models for Replica Synchronisation and Consistency in a Data Grid*. 10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10), San Francisco, California, USA, 2001.

[17] I. Foster, C. Kesselman, J. M. Nick, & S. Tuecke. The *Physiology of the Grid - An Open Grid Services Architecture for Distributed Systems Integration (Version: 6/22/2002)*. http://www.globus.org/research/papers/ogsa.pdf.

[18] I. Foster & C. Kesselman (eds.). *The Grid: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann: San Francisco, CA, 1999.

[19] L. Guy, P. Kunszt, E. Laure, H. Stockinger & K. Stockinger. *Replica Management in Data Grids.* Technical Report, GGF5 Working Draft. Edinburgh, Scotland, 2002.

[20] W. Hoschek, J. Jean-Martinez, A. Samar, H. Stockinger, K. Stockinger. *Data Management in an International Data Grid Project*. In Proceedings of the First IEEE/ACM International Workshop on Grid Computing R. Buyya and M. Baker, Eds. Lecture Notes In Computer Science, 1971, 77-90, Springer-Verlag, London, 2000.

[21] A. Keller, & A. Reinefeld. *Anatomy of a Resource Management System for HPC Clusters.* Annual Review of Scalable Computing, 3, 2001.

[22] G. Kola, T. Kosar, & M. Livny. *Run-time Adaptation of Grid Data Placement Jobs*. In Proceedings of Int. Workshop on Adaptive Grid Middleware (AGridM 2003), New Orleans, LA, USA, 2003.

[23] T. Kosar, & M. Livny. *Stork: Making Data Placement a First Class Citizen in the Grid*. In Proceedings of the 24th Int. Conference on Distributed Computing Systems, Tokyo, Japan, 2004.

[24] G. Krüger. *Go To Java 2 – Handbuch der Java-Programmierung*. Addision-Wesley, 2000.

[25] V. Kumar. *Algorithms for Constraint-Satisfaction Problems: A Survey*. AI Magazine. 13, 1 (1992), 32-44.

[26] Kommunikation – Übertragungsnetz: *Berliner Tagesbelastung im Vergleich*. Vattenfall Europe Information Services GmbH. Erhalten über Anfrage an Pressestelle.

[27] J. Maassen, R. V. van Nieuwpoort, T. Kielmann, K. Verstoep & M. den Burger. *Middleware Adaption with the Delphoi Service*. In Proceedings of AGridM 2004 Workshop on Adaptive Grid Middleware, Antibes Juan-les-Pins, France, 2004.

[28] R. V. v. Nieuwpoort, J. Maassen, K. Verstoep & T. Kielmann. *Implementation of Core Adaption Functionality.*
http://www.gridlab.org/Resources/Deliverables/D7.3new.pdf

[29] T. A. Pham. *Einbettung neuer Verwaltungsmethoden in die hierarchische Dateisystemsicht*. Diplomarbeit. Technische Universität Berlin, 2005.

[30] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. Dissertation. University of Wisconsin, Madison, Winconsin, USA. 2001.

[31] R. Raman, M. Livny & M. Solomon. *Matchmaking: Distributed Resource Management for High Throughput Computing*. In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, 1998.

[32] R. Raman, M. Livny, & M. Solomon. *Resource Management through Multilateral Matchmaking*. In Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00), 2000.

[33] A. Reinefeld & F. Schintke. *Concepts and Technologies for a Worldwide Grid Infrastracture*. Euro-Par 2002 Parallel Processing, Volume 2400 of Lecture Notes in Computer Science, Springer, 2400, 62-71, 2002.

[34] V. Ribeiro, R. Riedi, R. Baraniuk, J. Navratil & L. Cottrell. *pathChirp: Efficient Available Bandwidth Estimation for Network Paths.* In Passive and Active Measurement Workshop (PAM 2003). NLANR, 2003.

[35] T. Röblitz & A. Reinefeld. *Co-Reservation with the Concept of Virtual Ressources*. In Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid, Cardiff, Wales, UK, 2005.

[36] T. Röblitz, F. Schintke & A. Reinefeld. *Resource Reservations with Fuzzy Requests*. Concurrency and Computation: Practice and Experience (to appear), 2005.

[37] F. Schintke, T .Schütt, T. Ganjineh, M. Moser, T. Pham, C. v. Prollius & R. Tuschl. *ZIB-DMS: A Scalable Data Management System*. Poster. Zuse Institute Berlin, Germany, 2005.

[38] J. M. Schopf. *Ten Actions when Grid Scheduling: The User as a Grid Scheduler*. In Grid Resource Management: State of the Art and Future Trends, J. Nabrzyski, J. M. Schopf & J. Weglarz, Eds. Kluwer Academic Publishers, Norwell, MA, 15-23, 2004.

[39] M. v. Steen & A. Tannenbaum. *Verteilte Systeme, Grundlagen und Paradigmen*. Prentice Hall, 2003.

[40] T. Tannenbaum, D. Wright, K. Miller & M. Livny. *Condor – A Distributed Job Scheduler*. Beowulf Cluster Computing in Linux, MIT-Press, 2001.

[41] D. Thain, T. Tannenbaum, M. Livny. *Condor and the Grid*. In F. Berman, G. Fox, A. Hey (eds.). Grid Computing: Making the Global Infrastructure a Reality, John Wiley & Sons, 2003.

[42] M. A. Trick. *A Tutorial on Integer Programming*. http://mat.gsia.cmu.edu/orclass/integer/integer.html.

[43] A tutorial for programming the C++ implementation. http://www.cs.wisc.edu/condor/classad/c++tut.html.

[44] H. R. Varian. *Grundzüge der Mikroökonomik*. 5. Auflage, Oldenbourg Wissenschaftsverlag, München, 2001.

[45] S. Venugopal, R. Buyya & L. Winton. *A Grid Service Broker for Scheduling Distributed Data-Oriented Applications on Global Grids*. Technical Report, GRIDS-TR-2004-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, 2004.

[46] S. Venugopal, R. Buyya & L. Winton. *A Grid Service Broker for Scheduling e-Science Applications on Global Data Grids*. Journal of Concurrency and Computation: Practice and Experience, Wiley Press, USA (2005).

[47] J. Witte. *Implementierung einer NFS-Schnittstelle im Kontext eines Grid-Datamanagements-Systems*. Diplomarbeit. Humboldt-Universität zu Berlin, 2005.

[48] R. Wolski, N. Spring, & C. Peterson. *Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service*. In Proceedings of the SC97, 1997.

13 Anhang

13.1 Beispiel für eine Jobbeschreibung des Resourcebrokers

```
Executable = "Sample.class"                    // auszuführendes Programm
Universe = "java"                              // Ausführungsumgebung
parameters = "infile outfile"                  // Startparameter
In = "in"                                      // Datei für Eingabeumlenkung
out = "output"                                 // Datei für Ausgabeumlenkung
err = "errfile"                                // Datei für Fehlerumlenkung
Log = "loop.log"                               // Protokolldatei
Requirements = Machine.Memory >= 32 && Machine.OpSys == "IRIX65" && Machi-
ne.Arch =="SGI"                                // Anforderungen für einen Match
Rank = Machine.Memory                          // Rangwert für Sortierung der Matches
Image_Size = 28 Meg
Deadline = 2005/10/13 08:20:00                 // spätestes Jobende
MaxCosts = 4000                                // Maximalkosten für den Gesamtjob
Starttime = 2005/10/13 08:07:00                // früheste Startzeit
StartTimeFor ="JOB"                            // Startzeit gilt für den Job
Duration = 3000                                // Dauer des Jobs in Sekunden
Mincosts=true                                  // Kosten sollen minimiert werden
Mintime=true                                   // Zeit soll minimiert werden
transfer_input_files = "infile"                // zusätzliche zu transferiende Dateien
transfer_output_files = "outfile"              // zu transferierende Ausgabedateien nach dem Job
Initialdir = run_1                             // Verzeichnis zur Ablage von Dateien
Queue                                          // Kommando, eine JobClassAd zu erstellen
```

13.2 Erzeugte JobClassAd aus der Jobbeschreibung

```
[
  TYPE="JOB";                                  // Typ der ClassAd
  NAME="maggie.zib.de";                        // Name des absendenden Rechners
  PORT="1099";                                 // Port des Rechners
  UNIVERSE="java";
  LOCALPATH="/home/lichtens/.eclipse/ResBroker/bin/../sample";
                                               // Arbeitsverzeichnis beim Absender
  DATACOUNT=3;                                 // Anzahl der Elemente der Datafiles-Auflistung
  DATAFILES={                                  // Liste aller vor dem Job zu kopierenden Dateien
    [
      LABEL="DATAFILE0";                       // Name des Abschnitts
      KIND=EXEC;                               // Dateityp, hier: ausführbares Programm
      FILE=getFile("Sample.class", DATAFILE0, MINSTARTTIME,
        MAXENDTIME, parent.MACHINE.MACHINE.NAME);
                                               // Funktion, die Verfügbarkeit der Datei testet
      MAXENDTIME=parent.MACHINE.STARTTIME;
                                               // spätestes Ende des Datentransfers
      MINSTARTTIME=absTime("2005-10-13T04:12:42+02:00");
                                               // früheste Startzeit des Jobs (ggf. Absendezeit)
      STARTTIME=FILE.STARTTIME;                // Startzeit des Datentransfer (zur Kompatibilität)
      ENDTIME=FILE.ENDTIME;                    // Endzeit des Datentransfers (zur Kompatibilität)
      REQUIREMENTS=DATAFILE0.TYPE=="DATA" && FILE.isAvailable;
                                               // Anforderungen an die matchende DataClassAd
    ],
    [
      LABEL="DATAFILE1";
      KIND="IN";
      FILE=getFile("in", DATAFILE1, MINSTARTTIME, MAXENDTIME,
        parent.MACHINE.MACHINE.NAME);
```

```
      MAXENDTIME=parent.MACHINE.STARTTIME;
      MINSTARTTIME=absTime("2005-10-12T10:52:42+02:00");
      STARTTIME=FILE.STARTTIME;
      ENDTIME=FILE.ENDTIME;
      REQUIREMENTS=DATAFILE1.TYPE=="DATA" && FILE.isAvailable;
    ],
    [
      LABEL="DATAFILE2";
      KIND="DATA";
      FILE=getFile("infile", DATAFILE2, MINSTARTTIME, MAXENDTIME,
        parent.MACHINE.MACHINE.NAME);
      MAXENDTIME=parent.MACHINE.STARTTIME;
      MINSTARTTIME=absTime("2005-10-12T10:52:42+02:00");
      STARTTIME=FILE.STARTTIME;
      ENDTIME=FILE.ENDTIME;
      REQUIREMENTS=DATAFILE2.TYPE=="DATA" && FILE.isAvailable;
    ]
  }
  EXECUTABLE=DATAFILES[0].FILE.NAME;   // Verweis auf ausführbares Programm
  PARAMETERS="infile outfile";
  IN=DATAFILES[1].FILE.NAME;           // Verweis auf Eingabedatei
  OUT="output";
  ERR="errfile";
  LOG="loop.log";
  MACHINE=[
      REQUIREMENTS=Machine.Memory>=32 && Machine.OpSys=="IRIX65"
        && Machine.Arch=="SGI" && TIME.ISAVAILABLE;
                                       // Anforderungen an MachineClassAd
      RANK=Machine.Memory;
      COSTS=getCosts(MACHINE.NAME, STARTTIME, parent.DURATION);
                                       // Kosten für die Ausführung des Jobs
      TIME=checkmachine(machine.name, parent.MINSTARTTIME,
        parent.DURATION, parent.DEADLINE);
                                       // Zeitfenster für die Ausführung des Jobs
      STARTTIME=TIME.START;            // Startzeit des Jobs
      ENDTIME=TIME.END;                // Endzeit des Jobs
  ];
  IMAGE_SIZE=28;
  DEADLINE=absTime("2005-01-13T08:18:00+02:00");
  MAXCOSTS=4000;
  MINSTARTTIME=absTime("2005-01-13T08:05:00+02:00");
                                       // entspricht STARTTIME der Jobbeschreibung
  STARTTIMEFOR="JOB";
  DURATION=3000;
  MINCOSTS=true;
  MINTIME=true;
  TRANSFER_OUTPUT_FILES="outfile";
  INITIALDIR=run_1;
  COSTS=MACHINE.COSTS + DATAFILES[0].FILE.COSTS
    + DATAFILES[1].FILE.COSTS + DATAFILES[2].FILE.COSTS;
                                       // Gesamtkosten des Jobs
  REQUIREMENTS=COSTS<=MAXCOSTS && MACHINE.Endtime<=DEADLINE;
                                       // gloable Anforderungen
  RANK=(1/COSTS) * (1/dateToReal(MACHINE.Endtime)
    -minvalue(DATAFILES[0].FILE.Starttime,
    DATAFILES[1].FILE.Starttime, DATAFILES[2].FILE.Starttime));
                                       // globaler Rangwert
]
```

Grau markierte Zeilen wurden aus der Jobbeschreibung 1:1 übernommen, die anderen Zeilen wurden durch den J2B-Client auf Basis der Jobbeschreibung er-

stellt. Es ist ersichtlich, dass der Umfang stark zugenommen hat und dass die Erzeugung einer JobClassAd sehr komplex und daher dem Anwender nicht zuzumuten ist. Rote Zeilen kennzeichnen Aufrufe benutzerdefinierter Funktionen, die während des Matchmakings dynamische Informationen erfragen.

13.3 Beispiel für eine MachineClassAd

```
[
  Type="MACHINE";                    // Typ der ClassAd
  Arch="SGI";                        // Systemarchitektur der Ressource
  Memory=768;                        // Arbeitsspeicher des Ressource
  OpSys="IRIX65";                    // Betriebssystem
  REQUIREMENTS=true;                 // Anforderungen (hier: keine)
  RANK=0;                            // Rangwert (hier: keiner)
  NAME="maggie.zib.de"               // Name der Ressource
  PORT="1099";                       // Port der Ressource
]
```

Die ClassAd für Ressource ist absichtlich klein gehalten worden, da das Hauptaugenmerk auf der JobClassAd und den globalen Anforderungen lag.

13.4 Beispiel für eine DataClassAd

```
[
  TYPE="DATA";                       // Typ der ClassAd
  NAME="maggie.zib.de";              // Name des Rechners
  PORT="1099";                       // Port des Rechners
  REQUIREMENTS=true;                 // Anforderungen (hier: keine)
  RANK=0;                            // Rangwert (hier: keiner)
  DMSTYPE="ZIBDMS"                   // Typ des DMS
]
```

Die ClassAd für Ressourcen ist absichtlich klein gehalten worden, da das Hauptaugenmerk auf der JobClassAd und den globalen Anforderungen liegt.

13.5 MatchedClassAd aus den drei ClassAds

```
[
  JOB=[                              // erweiterte JobClassAd
    TYPE="JOB";
    NAME="maggie.zib.de";
    PORT="1099";
    UNIVERSE="java";
    LOCALPATH="/home/lichtens/.eclipse/ResBroker/bin/../sample";
    DATACOUNT=3;
    DATAFILES={
      [
        LABEL="DATAFILE0";
        KIND=EXEC;
        FILE=[
          NAME="Sample.class";
          STARTTIME=absTime("2005-10-12T11:00:29+02:00");
```

```
      ENDTIME=absTime("2005-10-12T11:00:29+02:00");
      DURATION=5;
      COSTS=10;
      ISAVAILABLE=true;
    ];
    MAXENDTIME=parent.MACHINE.STARTTIME;
    MINSTARTTIME=absTime("2005-10-12T10:59:42+02:00");
    STARTTIME=FILE.STARTTIME;
    ENDTIME=FILE.ENDTIME;
    REQUIREMENTS=((DATAFILE0.TYPE=="DATA")&&FILE.isAvailable);
    DATAFILE0=PARENT.DATAFILEPATH0;  // Verweis auf die zugehörige DataClassAd[50]
  ]; // JOB.DATAFILES[0]
  [
    LABEL="DATAFILE1";
    KIND="IN";
    FILE=[                          // Ergebnis der Funktion getFile(...)
      NAME="in";                    // lokaler Dateiname oder vollständiger Pfad
      STARTTIME=absTime("2005-10-12T11:00:29+02:00");
                                    // geplanter Transferstart
      ENDTIME=absTime("2005-10-12T11:00:29+02:00");
                                    // geplantes Transferende
      DURATION=5;                   // geschätzte Transferzeit
      COSTS=10;                     // Kosten für den Transfer
      ISAVAILABLE=true;             // besagt, dass die Datei verfügbar ist
    ];
    MAXENDTIME=parent.MACHINE.STARTTIME;
    MINSTARTTIME=absTime("2005-10-12T10:59:42+02:00");
    STARTTIME=FILE.STARTTIME;
    ENDTIME=FILE.ENDTIME;
    REQUIREMENTS=((DATAFILE1.TYPE=="DATA")&&FILE.isAvailable);
    DATAFILE1=PARENT.DATAFILEPATH1;

  ]; // JOB.DATAFILES[1]
  [
    LABEL="DATAFILE2";
    KIND="DATA";
    FILE=[
      NAME="infile";
      STARTTIME=absTime("2005-10-12T11:00:29+02:00");
      ENDTIME=absTime("2005-10-12T11:00:29+02:00");
      DURATION=5;
      COSTS=10;
      ISAVAILABLE=true
    ];
    MAXENDTIME=parent.MACHINE.STARTTIME;
    MINSTARTTIME=absTime("2005-10-12T10:59:42+02:00");
    STARTTIME=FILE.STARTTIME;
    ENDTIME=FILE.ENDTIME;
    REQUIREMENTS=((DATAFILE2.TYPE=="DATA")&&FILE.isAvailable);
    DATAFILE2=PARENT.DATAFILEPATH2;
  ]; // JOB.DATAFILES[2]
} // JOB.DATAFILES
EXECUTABLE=DATAFILE0.FILE.NAME;
PARAMETERS="infile outfile";
IN=DATAFILE1.FILE.NAME;
OUT="output";
ERR="errfile";
```

[50] Korrekter Verweis wäre DATAFILE0=PARENT.PARENT.DATAFILE[0]. Aufgrund eines Fehlers der Java-Implementation des ClassAd-API bei mehr als einem PARENT-Verweis in einem Attribut muss dieser Verweis auf zwei Attribute gesplittet werden.

```
  LOG="loop.log";
  MACHINE=[
    REQUIREMENTS=Machine.Memory>=32 && Machine.OpSys=="IRIX65"
      && Machine.Arch=="SGI" && TIME.ISAVAILABLE);
    RANK=Machine.Memory;
    COSTS=3000;                          // Ergebnis der Funktion getCosts(...)
    TIME=[                               // Ergebnis der Funktion checkMachine(...)
      START=absTime("2005-01-13T08:17:00+02:00");
      END=absTime("2005-01-13T08:17:00+02:00");
      ISAVAILABLE=true;
    ];
    STARTTIME=TIME.START;
    ENDTIME=TIME.END;
    MACHINE=PARENT.MACHINEPATH;          // Verweis auf die zugehörige MachineClassAd
  ]; // JOB.MACHINE
  IMAGE_SIZE=28;
  DEADLINE=absTime("2005-01-13T08:17:00+02:00");
  MAXCOSTS=4000;
  MINSTARTTIME=absTime("2005-01-13T08:17:00+02:00");
  STARTTIMEFOR="JOB";
  DURATION=3000;
  MINCOSTS=true;
  MINTIME=true;
  TRANSFER_OUTPUT_FILES="outfile";
  INITIALDIR=run_1;
  COSTS=MACHINE.COSTS + DATAFILES[0].FILE.COSTS
    + DATAFILES[1].FILE.COSTS + DATAFILES[2].FILE.COSTS;
  REQUIREMENTS=COSTS<=MAXCOSTS && MACHINE.Endtime<=DEADLINE;
  RANK=(1/COSTS) * (1/(dateToReal(MACHINE.Endtime)
    - minvalue(DATAFILES[0].FILE.Starttime,
    DATAFILES[1].FILE.Starttime, DATAFILES[2].FILE.Starttime));
  MACHINEPATH=PARENT.MACHINE;            // Verweise auf die zugehörigen ClassAds
  DATAFILEPATH0=PARENT.DATAFILE0;[50]
  DATAFILEPATH1=PARENT.DATAFILE1;
  DATAFILEPATH2=PARENT.DATAFILE2
];
MACHINE=[    // erweiterte MachineClassAd
  Arch="SGI";
  Memory=768;
  OpSys="IRIX65";
  REQUIREMENTS=true;
  RANK=0;
  NAME="maggie.zib.de";
  JOB=PARENT.JOB.MACHINE;                // Verweis auf die zugehörige JobClassAd
];
DATAFILE0=[ // erweiterte DataClassAd für die erste Datei
  TYPE="DATA";
  NAME="maggie.zib.de";
  PORT="1099";
  REQUIREMENTS=true;
  RANK=0;
  DMSTYPE="ZIBDMS";
  JOB=PARENT.JOB.DATAFILE2;              // Verweis auf die zugehörige JobClassAd
];
DATAFILE1=[
  TYPE="DATA";
  NAME="maggie.zib.de";
  PORT="1099";
  REQUIREMENTS=true;
  RANK=0;
  DMSTYPE="ZIBDMS";
  JOB=PARENT.JOB.DATAFILE2
```

```
  ];
  DATAFILE2=[
    TYPE="DATA";
    NAME="maggie.zib.de";
    PORT="1099";
    REQUIREMENTS=true;
    RANK=0;
    DMSTYPE="ZIBDMS";
    JOB=PARENT.JOB.DATAFILE2
  ]
]
```

Graue Zeilen wurden 1:1 aus den jeweiligen ClassAds übernommen. Grüne Zeilen zeigen die Ergebnisse der benutzerdefinierten Funktionen, die beim Matchmaking externe Komponenten abgefragt haben. Die Erweiterungen der MatchedClassAd fallen gering aus, hauptsächlich wurden Verweise auf die anderen ClassAds sowie die Ergebnisse der Funktionsaufrufe eingefügt.

13.6 Zusammenfassung des ClassAd-APIs

Dieser Abschnitt stellt die wichtigsten Klassen der Java-Implementation des ClassAd-APIs [C4] vor. ClassAds werden intern als Bäume repräsentiert, wobei die einzelne Ausdrücke Knoten oder Blätter eines Baumes sind. Ausdrücke können mit den folgenden Klassen repräsentiert werden:

Expr	Abstrakte Basisklasse für alle Ausdrücke, repräsentiert einen Knoten des Baumes
RecordExpr	Eine „normale“ ClassAd (Bsp.: **[a=1; b=2; c=5, …]**)
SelectExpr	Auswahl aus einer **RecordExpr** (Bsp.: **a.b**)
ListExpr	Eine Liste (Bsp: **{[a=3], 5, "test"}**)
SubscriptExpr	Auswahl eines Listenelements (Bsp.: **a[0]**)
CondExpr	Eine Auswahlbedingung (Bsp.: **a ? b : c**)
Op	Binärer oder unärer Operator
FuncCall	Funktionsaufruf
AttrRef	Eine Attributreferenz im Sinne eines Identifiers
Constant	eine Konstante (etwa eine Zeichenkette, eine Zahl oder einer der Werte **TRUE**, **FALSE**, **ERROR** oder **UNDEFINED**)

AttrRef und **Constant** sind können Blätter im Ausdrucksbaum (*expression tree*) sein. **Constant** kann auch interner Knoten sein. Alle anderen Typen können nur interne Knoten repräsentieren.

Jeder Ausdruck kann mittels der Funktion **eval()** ausgewertet werden. **eval(Env env)** sucht zusätzlich in der Umgebung **env**, welche einen Stapel mit **RecordExpr**s enthält, der für die Auswertung mit durchsucht werden kann.

ClassAdParser ist eine Klasse zum Parsen einer ClassAd aus einer Zeichenkette oder einem XML-Dokument. Das folgende Beispiel erstellt aus dem String **s** eine ClassAd:

```
String s = "[a=1, b=2, c=d, d=a]"
ClassAdParser parser = new ClassAdParser(s);
Expr expr = parser.parse();
```

Beispiel zum Hinzufügen von Attributen in eine **RecordExpr**:

```
int i = 3;
RecordExpr re = (RecordExpr)expr;
re.insertAttribute("zahl", Constant.getInstance(i));
```

Constant.getInstance erzeugt Konstante vom jeweiligen Typ.

Die Klasse **ClassAd** dient der ClassAd-Auswertung:

bind(RecordExpr r1, RecordExpr2 r2)	Verbindet beide **RecordExpr**s in einer neuen ClassAd und erzeugt jeweils die Attribute **self** und **other**.
eval(RecordExpr re, String attr)	Wertet das angegebene Attribut im Kontext der ClassAd aus. Die Umgebung wird hierbei automatisch erstellt.
match(Expr e1, Expr e2)	Matcht die beiden Ausdrücke. Zunächst werden beide mit **bind** zusammengefügt. Anschließend werden **e1.Requirements** und **e2.Requirements** ausgewertet. Ergeben beide den Wert „wahr“, so werden **e1.Rank** und **e2.Rank** berechnet und zurückgegeben.