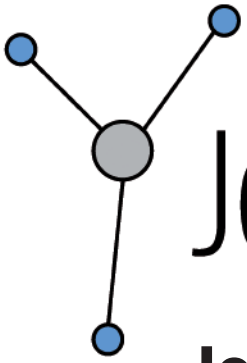
Matthias Bonn



JoSchKa

Jobverteilung in heterogenen und unzuverlässigen Umgebungen

universitätsverlag karlsruhe

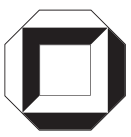
Matthias Bonn

JoSchKa
Jobverteilung in heterogenen und unzuverlässigen Umgebungen

JoSchKa

Jobverteilung in heterogenen und unzuverlässigen Umgebungen

von
Matthias Bonn



universitätsverlag karlsruhe

Dissertation, Universität Karlsruhe (TH)
Fakultät für Wirtschaftswissenschaften, 2008
Tag der mündlichen Prüfung: 21.07.2008
Referent: Prof. Dr. H. Schmeck
Korreferent: Prof. Dr. W. Juling

Impressum

Universitätsverlag Karlsruhe
c/o Universitätsbibliothek
Straße am Forum 2
D-76131 Karlsruhe
www.uvka.de

Universitätsverlag Karlsruhe 2008
Print on Demand

ISBN: 978-3-86644-288-7

Für Sventje und Eva

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) der Universität Karlsruhe (TH). Vor allem meine Aufgabe als Rechnerbeauftragter der Forschungsgruppe „Effiziente Algorithmen“ führte zur Entwicklung der ersten Versionen des hier beschriebenen Jobverteilsystems.

Mein Dank gilt meinem Doktorvater Herrn Prof. Dr. Hartmut Schmeck, der mir während der Arbeit an JobSchKa und der daraus entstandenen Dissertation immer das Gefühl gab, an etwas Nützlichem und Wichtigem zu arbeiten.

Weiterhin danke ich meinen Kollegen Andreas Kamper, Lukas König, Holger Prothmann, Urban Richter, Bernd Scheuermann und Christian Schmidt, die meine Software intensiv nutz(t)en, für ihre Leidensfähigkeit und die vielen wertvollen Tipps und Anregungen zur Verbesserung von JobSchKa. Ein weiterer Dank gilt ihnen für viele nützliche Diskussionen jeglicher Art, für den Antrieb, die Entwicklungsumgebung gegen die Textverarbeitung zu tauschen, und letztlich für das hilfreiche Korrekturlesen der Arbeit. Frederic Toussaint und dem CIP-Pool-Team danke ich für die Bereitschaft, den Rechnerpool für Tests und letztlich zum dauerhaften Produktivbetrieb zur Verfügung zu stellen.

Außerdem danke ich meinen Eltern, die mich immer unterstützt haben, sowie meiner Familie. Insbesondere meine Frau Sventje motivierte mich immer wieder, die Dissertation zu Ende zu schreiben.

Ihr und meiner Tochter Eva widme ich diese Arbeit.

Matthias Bonn

Karlsruhe, im September 2008

Inhalt

Abbildungen

Kapitel 1

Einführung

Die Nutzung von Personal Computern (PCs) und Internet hat sich im Laufe der letzten zehn Jahre sowohl im betrieblichen Arbeitsumfeld als auch im privaten Bereich weit verbreitet. Neben dem nicht mehr wegzudenkenden Einsatz im geschäftlichen und universitären Umfeld werden PCs mit Internetanschluss im Heimbereich hauptsächlich für Dienste wie dem Word Wide Web oder für Email bzw. zur Unterhaltung durch Filme oder Spiele verwendet. Im universitären Umfeld dienen Pools aus Standard-PCs – neben dem Gebrauch als Arbeitsplatzrechner – dazu, die Studierenden z. B. bei Literaturrecherchen, Textverarbeitung oder Programmierübungen zu unterstützen.

Beim Erwerb eines derartigen Rechners erhält man heute für rund 1000 Euro eine moderne Multikern-CPU[1], die mit 2–3 Gigahertz Takt arbeitet, und 2–4 Gigabyte Hauptspeicher. Damit verfügt ein normaler Arbeitsplatzrechner heute über die Leistung, die vor wenigen Jahren ausschließlich Supercomputern vorbehalten war: Die Gleitkomma-Rechenleistung eines gut ausgestatteten PCs erreicht heute Größenordnungen, für die vor gut einem Jahrzehnt noch hunderte von Prozessoren parallel arbeiten mussten [Int08, DMS99]. Diese Performanz benötigen die PCs im alltäglichen Gebrauch, vor allem wenn sie für Video-, Multimedia-Anwendungen oder Spiele genutzt werden. Ist der Rechner jedoch mit der Darstellung von Webseiten oder dem Herunterladen von Emails beschäftigt, verbringen derartige Prozessoren die Zeit überwiegend damit, auf Nutzereingaben zu warten (siehe Abschnitt 1.1).

Im Zuge der fortschreitenden Verbreitung des Internets seit dem Ende der 1990er Jahre sind einige Projekte entstanden, die diese weltweit brachliegende Rechenleistung für wissenschaftliche Zwecke nutzen, beispielsweise SETI@home: Der Besitzer eines beliebigen Internetrechners installiert sich einen speziellen Bildschirmschoner, so dass die Maschine genau dann, wenn keine weitere Interaktivität stattfindet, Teleskopdaten zur Suche nach außerirdischem Leben analysiert. Hierbei handelt es sich um ein spezialisiertes Projekt, das nicht benötigte Rechnerleistung zu genau einem Zweck verwendet. Ziel der vorliegenden Arbeit soll es jedoch sein, eben diese CPU-Ressourcen von privat oder im universitären Umfeld eingesetzten Arbeitsplatzrechnern für verschiedenste, wissenschaftliche Untersuchungen unter Verwendung von Internet-Standards bestmöglich nutzbar zu machen, wenn z. B. der Einsatz eines Höchstleistungsrechners nicht möglich ist.

[1] Bei einer solchen CPU (Central Processing Unit) handelt es sich um ein Prozessormodul, bei dem mehrere (meist 2 oder 4) vollwertige, parallel arbeitende Einzelprozessoren in einem Chip integriert sind.

1.1 Motivation

Ein typischer aktueller Arbeitsplatzrechner verfügt über eine Rechenkapazität von mehreren Milliarden (numerischen) Operationen pro Sekunde, die ihm allerdings in vielen typischen Anwendungsfällen nicht abverlangt wird. Abb. 1.1 zeigt im linken Diagramm die CPU- und die Hauptspeicherauslastung eines Bürorechners an einem intensiven Arbeitstag, gemessen über einen Zeitraum von fünf Stunden. Trotz umfangreicher interaktiver Nutzung einer Entwicklungsumgebung und eines Verkehrssimulationsprogramms war der Prozessor des Notebooks zu mehr als drei Vierteln der Zeit nicht beschäftigt.

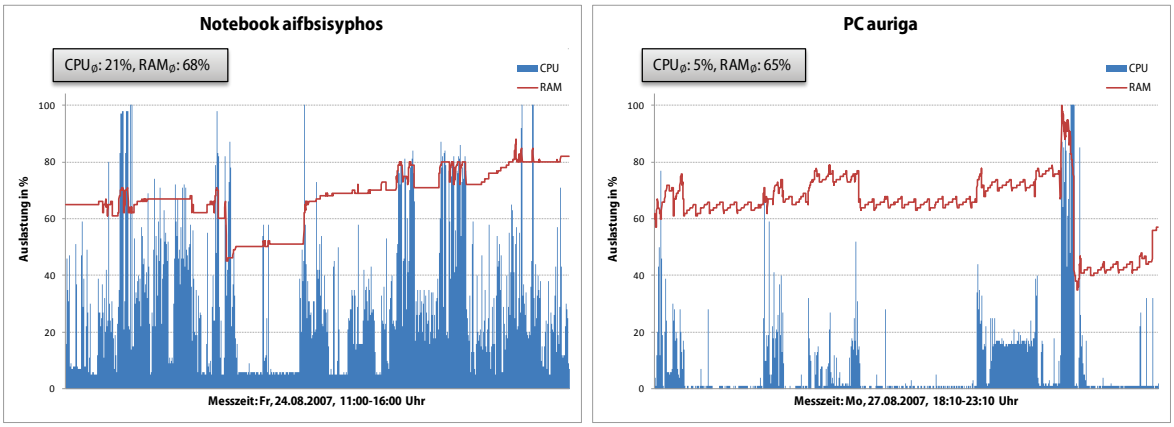


Abb. 1.1 Typische Rechnerbelastung

Das rechte Diagramm verdeutlicht noch stärker, wie unterbeschäftigt Rechner heutzutage sein können. Der Prozessor des dort gezeigten PCs, der privat und überwiegend für Email und den Abruf von Webseiten genutzt wurde, befand sich praktisch die ganze Zeit über im Leerlauf. Bei beiden aufgezeichneten Rechnern handelt es sich um ältere Singlekern-Systeme. Kämen moderne Multikern-Prozessoren zum Einsatz, würde sich die Rechnerlast nochmals halbieren oder gar vierteln.

Demgegenüber wird gerade in der universitären Forschung immer wieder dauerhaft oder kurzfristig hohe Rechenleistung benötigt [Sch07, RiMn08]. Häufig werden dabei zufallsgesteuerte Optimierungsverfahren entwickelt, die hunderte oder tausende Male auf dem gleichen Problem simuliert werden müssen, um sinnvolle Aussagen über ihre Leistungsfähigkeit und die ideale Parametrisierung machen zu können und sich deshalb hervorragend parallelisieren lassen. Für die permanente Nutzung stehen für derartige wissenschaftliche Simulationen häufig Hochleistungsrechner oder zumindest spezialisierte Rechencluster zur Verfügung, an deren Einsatz aber meist bestimmte Randbedingungen geknüpft sind. So ist man beispielsweise bei der Wahl der Programmiersprache oder des Betriebssystems eingeschränkt [SCCK08a] oder muss sich eventuell vorhandenen Prioritäten bei der Zuteilung der begrenzten Rechenkapazität unterordnen [SCCK08b]. Problematisch wird das beispielsweise für einen Anwender, der kurzfristig ein paar hundert Jobs (ein einzelner abgeschlossener Durchlauf eines Programms) durchführen muss und zu einem Großrechner keine Zugangsmöglichkeit hat oder eine dort nicht ausführbare Programmiersprache verwendet. Möglicherweise benötigt ein solcher Nutzer die Performanz eines derartigen Höchstleistungsrechners jedoch gar nicht, sondern es würden

schon einige der ohnehin zahlreich vorhandenen Standard-PCs ausreichen. Aber wie können solche Rechner mit möglichst geringem Aufwand nutzbar gemacht werden? Und wie kann man sie dann mit minimalem Einarbeitungsaufwand einsetzen?

Typisch für Systeme zur Verteilung und Ausführung rechenintensiver Aufgaben sind zwei Fälle: Sie sind entweder dafür gemacht, ad hoc nutzbar zu sein, oder um ad hoc Rechenleistung eines beliebigen Rechners verfügbar zu machen. Höchstleistungsrechner und die dort installierten Batchsysteme lassen sich – zumindest für erfahrene Daueranwender solcher Systeme – relativ einfach nutzen: Man entwickelt Anwendungen von vornherein für genau dieses Rechensystem, stellt sicher, dass alle benötigten Dateien vorhanden sind, übergibt die auszuführende Befehlszeile an das System und holt sich die Ergebnisse nach der Ausführung ab [SCCK08a]. Den Dateitransfer und die Remoteausführung der Befehlszeile auf einem der Rechenknoten übernimmt das Verteilsystem bzw. das darunterliegende Netzwerkdateisystem. Jedoch ist es nicht ohne einen gewissen Aufwand machbar – wenn nicht sogar unmöglich – einen beliebigen Arbeitsplatzrechner oder einen zur interaktiven Nutzung vorgesehenen Pool in solch einen Verbund zu integrieren. Die Tatsache, dass die Rechenknoten in ein Netzwerkdateisystem eingebunden sind und remote ansprechbar sein müssen, macht z. B. die Integration eines Internetrechners utopisch. Demgegenüber stehen Systeme, die speziell dafür ausgelegt sind, beliebige, auch im Internet verteilte Rechner einfach und schnell in einen Rechnerverbund aufnehmen zu können, z. B. in Form eines Bildschirmschoners. Solche Systeme lassen sich jedoch vom Entwickler einer parallelen Anwendung nur mit einem nicht unbeträchtlichen Aufwand einsetzen, da die zu verteilende Anwendung an beispielsweise genau diesen einen Bildschirmschoner angepasst werden muss. Meist bedarf es auch der Bereitstellung einer nicht unerheblichen Serverinfrastruktur [Boi07].

Wie lassen sich nun beide Eigenschaften in einem einzelnen System realisieren? Wie kann erreicht werden, dass der Nutzer eines Verteilsystems bei Bedarf ohne viel Einarbeitungszeit seine rechenintensiven Jobs auf andere Rechner verteilen lassen kann? Wie kann man ein System zur Verfügung stellen, bei dem gleichzeitig mit wenig administrativem Aufwand jeder beliebige Rechner – egal ob Hochleistungsserver oder Heimarbeitsplatz-PC – seine brachliegende Rechenleistung wissenschaftlichen Zwecken zur Verfügung stellen kann? Und wie lassen sich gegebenenfalls die auf diese Weise entstehenden heterogenen Rechenknoten mit ihren unterschiedlichsten Leistungs- und Zuverlässigkeitscharakteristika optimal mit Arbeit versorgen?

Diese Fragestellungen zu untersuchen, Lösungsansätze zu entwickeln und durch eine prototypische, jedoch voll praxistaugliche Implementierung zu realisieren ist die Aufgabe der vorliegenden Arbeit.

1.2 Ergebnisse

Mit JoSchKa (Job Scheduling Karlsruhe) wurde ein System zur Verteilung von rechenintensiven Jobs auf Standard-PCs konzipiert und umgesetzt, das ausschließlich standardisierte Internetprotokolle und Datenformate verwendet. Diese Eigenschaft und seine prinzipielle Arbeitsweise, bei der die Rechenknoten selbsttätig Jobs vom Server anfordern,

machen das gesamte System tauglich für den problemlosen Einsatz im Internet. JoSchKa zeichnet sich durch Flexibilität bezüglich der Art der Jobs, der unterstützten Programmiersprachen und Laufzeitumgebungen aus, und ist darüber hinaus für Entwickler paralleler Anwendungen schnell und unkompliziert einsetzbar. Der administrative Aufwand, einen normalen Arbeitsplatz- oder Heim-PC als Rechenknoten in das System zu integrieren, ist minimal.

Die Rechner, auf die JoSchKa die Jobs verteilt, weisen potentiell eine hohe Heterogenität auf, vor allem in Bezug auf die Art und Weise, wie zuverlässig sie jeweils arbeiten. Eine wesentliche Eigenschaft von JoSchKa besteht in der Analyse dieser Rechnerlandschaft, und in der Anpassung der Verteilungsstrategien an eben diese Beobachtungen. Unterscheiden sich die verfügbaren Jobs auf geeignete Weise in ihren Anforderungen, ordnet ein Scheduler sie so den vorhandenen Knoten zu, dass die durch Rechnerausfälle unnütz verbrauchte Rechenkapazität verringert wird. Das Konzept, die Verteilung der Jobs dem gemessenen Verhalten der Rechenknoten – mit dem Ziel der Leistungssteigerung – anzupassen, findet sich bei keinem der mit JoSchKa vergleichbaren und in der Praxis eingesetzten Verteilsysteme.

Zusätzlich wurde ein für Anwender leicht zu bedienendes graphisches Tool realisiert, was JoSchKa ebenfalls von anderen, rein kommandozeilenorientierten, Systemen unterscheidet. Des Weiteren wurde eine Programmierschnittstelle entwickelt, die die Einsatzbreite um die Möglichkeit einer transparenten parallelen Verteilung von objektorientiertem Programmcode erweitert.

1.3 Überblick

Im Kapitel 2 werden zunächst die Grundlagen der Internetkommunikation erläutert, jedoch beschränkt auf den Detailgrad, der für das technische Verständnis der Arbeit und des realisierten Verteilsystems bzw. seiner Arbeitsweise erforderlich ist. Dabei werden die wichtigsten Internetprotokolle, Middleware für beliebige verteilte Anwendungen sowie Techniken speziell zur Entwicklung verteilter paralleler Anwendungen vorgestellt. Der Hauptteil des Kapitels beschreibt die Konzepte des Höchstleistungs- bzw. Clustercomputing, des Gridcomputing und des verteilten Internet-basierten Rechnens auf Arbeitsplatz-Rechnern und grenzt dann diese voneinander ab, um die Einordnung der entwickelten Software in letzteres Konzept zu ermöglichen. Es werden mehrere typische Systeme zum verteilten Rechnen vorgestellt, wobei auf die Systeme *Condor*, *BOINC* und *Alchemi* detailliert eingegangen wird, da sie explizit dafür entworfen wurden, rechenintensive Jobs auch auf gewöhnlichen (fremden) Arbeitsplatzrechnern auszuführen und damit vergleichbare Ziele verfolgen wie JoSchKa.

Im Kapitel 3 werden zunächst die prinzipielle Arbeitsweise von JoSchKa und seine Client/Server-Architektur vorgestellt, indem das verwendete Kommunikationsmodell mit Webservices und die weiteren Schnittstellen sowie die beteiligten Teilsysteme und das Jobdatenmodell beschrieben werden. Ein zentraler Abschnitt widmet sich dem Server und seiner Kernkomponente. Deren Aufgabe besteht darin, das bisherige Verhalten der Rechenknoten zu beobachten, um mit diesem Wissen Schlüsse auf das zukünftige Verhal-

ten zu ziehen und die zu verteilenden Rechenjobs den Knoten entsprechend zuzuteilen. Auf diese Weise soll unnütz verbrauchte Rechenzeit durch ausfallende Rechner minimiert und die zu bearbeitenden Jobs so früh wie möglich fertiggestellt werden. Das Verfahren soll ferner alle konkurrierenden Benutzer möglichst „gerecht“ behandeln. Im Rahmen der Entwicklung geeigneter Verteilstrategien wurde ein Simulator entwickelt, der einzelne Rechenknoten mit unterschiedlichstem Leistungs- und Zuverlässigkeitsverhalten sowie unterschiedlichste Nutzerjobs verschiedener Laufzeit simulieren kann. Unter diesen reproduzierbaren Bedingungen wurden mehrere Verteilverfahren entworfen und nach verschiedenen Kriterien bewertet. Die erfolgversprechendsten davon wurden dann zu einem in der Praxis eingesetzten Scheduling-Algorithmus zusammengefügt. Im weiteren Verlauf von Kapitel 3 wird das auf den Rechenknoten laufende Teilsystem (Agent) und seine grundsätzliche Arbeitsweise bei der Ausführung eines Jobs beschrieben. Nach einem Abschnitt über Sicherheitsfragen folgt eine detaillierte Beschreibung der Anwendung und der Schnittstellen für Nutzer des Systems, wobei nicht nur ein graphisches Verwaltungstool für Batchjobs, sondern auch eine Programmierschnittstelle vorgestellt wird. Sie ermöglicht es, objektorientierte Codeblöcke einer parallelen Anwendung programmgesteuert von JoSchKa ausführen zu lassen.

Kapitel 4 gibt einen Einblick in die interne Strukturierung des Programmcodes des Servers und der Agenten und beschreibt anschließend detailliert die Installation und den Betrieb der beiden Systemkomponenten. Der Fokus liegt dabei nicht auf der Nutzung des Systems sondern auf den notwendigen administrativen Tätigkeiten, die auszuführen sind, wenn ein Server aufgesetzt bzw. ein Rechner/Rechnerpool zur Nutzung mit JoSchKa verfügbar gemacht werden soll und darauf, wie vorhandene Arbeitsplatz- oder Heimrechner zur Ausführung von Rechenjobs vorbereitet werden können. Darüberhinaus geht es der Frage nach, inwieweit die Hintergrundnutzung eines Arbeitsplatzrechners durch Rechenjobs dessen reguläre Nutzung stören kann.

Das abschließende Kapitel 5 fasst die Ergebnisse der Arbeit zusammen und widmet sich dem realen Dauereinsatz des Systems in einem rund 100 Knoten umfassenden Pool der Universität Karlsruhe, der durch einige dedizierte leistungsfähige Rechenserver ergänzt wurde. Abschließend wird ein Ausblick auf weitergehende Entwicklungen und perspektivische Funktionalitäten von JoSchKa gegeben.

Kapitel 2

Stand der Technik

Dieses Kapitel soll zunächst einen Überblick über die wichtigsten Aspekte der Computerkommunikation geben, soweit sie für das Verständnis der Arbeit notwendig sind. Dabei werden zunächst die prinzipiellen Architekturen verteilter Systeme im Internet vorgestellt, bevor auf die Grundlagen der Internetkommunikation eingegangen wird. Im Weiteren werden spezielle Techniken zum verteilten parallelen Rechnen erläutert und konkrete Systeme zur Jobverteilung vorgestellt, wobei zwischen Höchstleistungsrechnern und Systemen zur Jobverteilung auf Standard-PCs unterschieden wird. Ein Abschnitt über die Konzepte des Gridcomputing schließt das Kapitel ab.

2.1 Architekturen verteilter Systeme

Bei verteilten Systemen können prinzipiell zwei grundsätzliche Architekturen zum Einsatz kommen: Client/Server und Peer-to-Peer. Je nach System existieren auch beliebige Mischformen, jedoch sollen hier zunächst die beiden Extremformen unterschieden werden, auf die hybriden Ausprägungen wird dann jeweils bei der Erläuterung eines konkreten Systems (Abschnitte 2.3 und 2.4) eingegangen.

2.1.1 Client/Server

Das generelle Merkmal eines Client/Server-Systems liegt darin, dass es immer mindestens zwei Typen von Rechnern gibt: Der Server bietet einen oder mehrere Dienste an und die Clients nutzen diese Dienste. Die wesentliche strukturelle Eigenschaft eines solchen Systems besteht darin, dass die Clients ausschließlich mit dem Server kommunizieren, untereinander kennen sie sich nicht (Abb. 2.1).

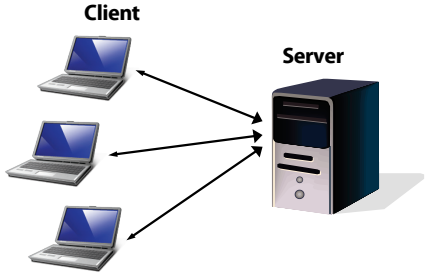


Abb. 2.1 Client/Server Architektur

Der Server verhält sich dabei immer passiv: Er bietet den Dienst an, aber erst wenn ein Client aktiv den Dienst beansprucht, wird er tätig. Ein typisches Beispiel für ein reines Client/Server-System wäre das World Wide Web (WWW) [TavS03]: Die Server bieten Webseiten oder andere Dokumente an, liefern diese aber erst dann an Clientrechner aus, wenn der dort laufende Webbrowser aktiv nach einer bestimmten Seite auf einem bestimmten Server fragt. Ein entscheidendes Merkmal bei Client/Server-Systemen ist somit der Verbindungsaufbau: Er wird immer vom Client initiiert (vgl. 2.2.1), was dazu führt, dass diese Architektur bei Internetdiensten für den Endanwender die mit Abstand am weitesten verbreitete ist. Die häufigsten Internetanwendungen sind das WWW und der Versand von Emails bzw. das Abfragen von Email-Accounts, beide arbeiten nach dem Client/Server-Prinzip. Auch andere populäre Internetanwendungen wie z. B. Instant Messaging oder der Internet Relay Chat (IRC) sind Client/Server-Systeme.

2.1.2 Peer-to-Peer (P2P)

Die Unterscheidung der beteiligten Rechner in Dienstgeber und Dienstnehmer existiert bei P2P-Netzen nicht. Dort tritt jeder beteiligte Knoten sowohl als Server als auch als Client auf (engl. *peer*, 'Gleichgestellter') und es bestehen direkte Verbindungen zwischen den Teilnehmern (Abb. 2.2). Eine typische Anwendung ist das Verteilen großer Datenmengen: Jeder Knoten besitzt Dateien oder Teile davon, die andere Knoten von ihm herunterladen können. Ein leistungsstarker zentraler Server mit schneller Internetanbindung ist dabei nicht notwendig. Ein funktionierendes P2P-Netz ist deswegen sehr robust gegen Rechnerausfälle: Verlässt ein Peer das Netz, ist das nicht weiter schlimm, es sind in aller Regel genügend andere Rechner vorhanden, die den Dienst (z. B. ein Dokument) anbieten können. Weiterhin besteht bei P2P-Netzen das Potential der guten Skalierbarkeit: Die Menge der Teilnehmer hat wenig Einfluss auf die Performance, während ein zentraler Server – abhängig von der Zahl der Clients – immer einen möglichen Engpass darstellt, der früher oder später an seine Kapazitätsgrenze gelangt.

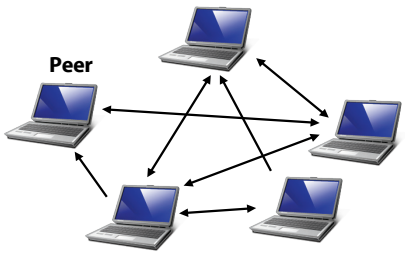


Abb. 2.2 Peer-to-Peer Architektur

Das Fehlen eines zentralen Servers erfordert allerdings Such- und Verbindungsverfahren, die weit aufwändiger sind als bei Client/Server-Systemen: Die Rechner kennen sich zunächst nicht und wissen auch nicht, welcher Knoten welche Daten besitzt. Aus diesem Grund existiert bei P2P-Netzen häufig ein virtuelles, darüber liegendes Netz, das die angebotenen Daten bzw. Dienste und die beteiligten Knoten (-Adressen) virtualisiert und von den realen Gegebenheiten abstrahiert. Beispiele für solche Overlay-Netze (engl. *over-*

lay, 'Überlagerung') sind das JXTA-Framework (auf das hier nicht näher eingegangen werden soll) oder eine MPI-Bibliothek (vgl. 2.2.3). Eine weitere Schwierigkeit besteht im Verbindungsaufbau: Jeder Peer muss direkt erreichbar sein, was den Betrieb im Internet erschwert (vgl. 2.2.1). Typische Anwendungen im Internet, die nach dem P2P-Prinzip arbeiten, sind Dateitauschbörsen wie z. B. BitTorrent oder Gnutella.

Viele P2P-Systeme sind als hybrides System ausgelegt, bei dem zentrales Wissen über die angebotenen Dienste und Knoten existiert, die eigentlichen Datentransfers aber direkt von Peer zu Peer abgewickelt werden. Ein Beispiel ist der nicht mehr existierende Musiktauschdienst Napster, bei dem ein zentraler Server die Knoten und die von ihnen angebotenen Daten kennt, die Dateitransfers aber direkt von Knoten zu Knoten durchgeführt werden. Ein weiteres Beispiel ist das SMTP-Mailsystem: Die DNS-Server (siehe 2.2.1, IP) im Internet wissen, welcher Mailserver für welche Domain zuständig ist, aber die eigentlichen Mails leiten die Mailserver untereinander direkt weiter. Auch Höchstleistungsrechner (vgl. 2.3) lassen sich als hybrides P2P-System betrachten: Ein zentrales System verteilt die Jobs an die Knoten, welche jedoch direkt untereinander gemeinsam einen parallelen Algorithmus durchführen.

2.2 Basistechnologien

2.2.1 Internet-basierte Datenübertragung/Kommunikation

Das im Rahmen dieser Arbeit entstandene Jobverteilungssystem (und insbesondere die verwendeten Kommunikationsmechanismen) basiert auf Internettechnologie [Tan97]. Aus diesem Grund werden zunächst kurz die wesentlichen Merkmale und die grundlegende Funktionsweise des Internets dargestellt, bevor auf spezielle Internet-basierte Standardanwendungsprotokolle eingegangen wird.

Die gesamte Kommunikation im Internet wird über die TCP/IP-Protokollfamilie durchgeführt, die zum einen von der physikalischen Datenübertragung abstrahiert, und zum anderen eine fest definierte Schnittstelle bietet, mit der beliebige Anwendungen das Internet zur Kommunikation nutzen können. Abb. 2.3 zeigt vereinfacht die Arbeitsweise der Internetkommunikation: Eine Anwendung auf Rechner A (z. B. ein Webbrowser) möchte mit einer Anwendung auf Rechner B (z. B. einem Webserver) kommunizieren. Die Nutzdaten werden durch die einzelnen Schichten durchgereicht und verarbeitet, wobei jede Schicht die Daten der darüber liegenden verpackt und mit eigenen Protokollinformationen versieht. Das gesamte Datenpaket wird dann über das Netzwerk physikalisch direkt oder mit mehreren Zwischenstationen an den Rechner B übertragen, wo die Daten die einzelnen Schichten in umgekehrter Reihenfolge durchlaufen, bis die Nutzdaten an der Zielanwendung angekommen sind. Die bei allen Rechnern im Internet vorhandenen Protokolle sind dabei das Transportprotokoll TCP und das Vermittlungsprotokoll IP, welche im Folgenden kurz erläutert werden. Auf die darunterliegenden Protokolle und physikalischen Übertragungsmechanismen wird jedoch nicht weiter eingegangen, denn sie sind für die vorliegende Arbeit nicht weiter relevant.

IP

Das Internet Protocol IP [RFC791] bildet die Schnittstelle zur eigentlichen Netzwerkübertragungstechnik (z. B. Ethernet) und ist dafür zuständig, dass die einzelnen Datagramme – das Internet arbeitet paketvermittelnd – den Weg vom Absender zum Empfänger durch das Netz finden. Die Pakete müssen dabei nicht notwendigerweise alle den gleichen Weg nehmen und können in geänderter Reihenfolge beim Empfängerrechner ankommen. Die Rechnernamen selbst werden vom Domain Name System (DNS) [RFC1034] in 32 oder 128 Bit breite (je nach IP-Version) numerische Adressen übersetzt. Die eigentliche protokollinterne Adressierung erfolgt dann mit diesen numerischen IP-Adressen.

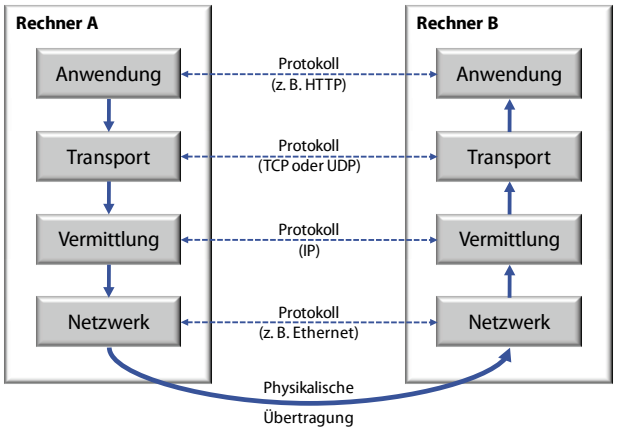


Abb. 2.3 Kommunikation im Internet

TCP

Das über der IP-Schicht arbeitende Transmission Control Protocol TCP [RFC793] stellt die reihenfolgetreue Auslieferung der Nutzdaten an die Anwendung sicher und sorgt mit Quittierungs- und Sendewiederholungsmechanismen für eine zuverlässige Verbindung. Eine TCP-Verbindung besteht nur zwischen den beiden beteiligten Endsystemen, es handelt sich dabei also um eine virtuelle Verbindung. Eine weitere wichtige Funktion des Protokolls ist die Bereitstellung einer Anwendungsschnittstelle, mit der jede beliebige Anwendung (z. B. ein Emailclient) eine TCP-Verbindung zu einer anderen Anwendung auf einem anderen Rechner (z. B. einem Mailserver) öffnen kann. Der Endpunkt einer solchen Verbindung wird als Socket bezeichnet und besteht aus der IP-Adresse des jeweiligen Rechners und einer 16 Bit breiten Portnummer, die der Adressierung der Anwendung auf dem Rechner dient.

Ein für diese Arbeit wichtiger Aspekt ist der TCP-Verbindungsaufbau: Für das erfolgreiche Zustandekommen einer Verbindung spielt es oft eine Rolle, welcher der beiden beteiligten Rechner den Aufbau initiiert. Viele Router im Internet, vor allem die in den im Heimbereich eingesetzten DSL-Modems integrierten, haben die Eigenschaft, ein lokales Netzwerk vom Internet auf der TCP/IP-Ebene zu trennen. Ein hinter einem solchen Router versteckter Rechner kann vom Internet aus nicht mit einem TCP-Verbindungswunsch kontaktiert werden, er ist quasi unsichtbar. Dieses als Network Address Translation (NAT) [RFC1631] bekannte Verhalten bietet den Vorteil, dass sich mehrere lokale Rech-

ner eine gemeinsame global eindeutige IP-Adresse teilen können, und dass sie vor Zugriffen aus dem Netz geschützt sind. Läuft der Verbindungsaufbau jedoch in die andere Richtung (also vom lokalen Netz ins Internet), kann jeder Internetdienst genutzt werden. Ist die Verbindung nämlich einmal von innen nach außen geöffnet worden, erkennt der Router an den Portnummern der Pakete, an welchen der lokalen Rechner er sie weiterleiten muss, und die Verbindung kann in beide Richtungen genutzt werden (Abb. 2.4). Als Folge davon funktionieren Client/Server-Architekturen in NAT-Umgebungen meist problemlos, was bei Peer-2-Peer Systemen nicht unbedingt gegeben sein muss.

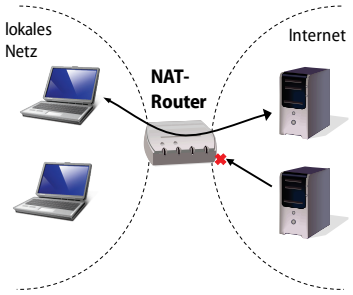


Abb. 2.4 Network Address Translation

Die Tatsache, dass eine TCP-Verbindung durch den Router hindurch nur in eine Richtung aufgebaut werden kann, spielt eine entscheidende Rolle bei der Wahl der Architektur des in dieser Arbeit vorgestellten Jobverteilungssystems. Zunächst sollen jedoch noch einige typische Anwendungsprotokolle, die auf TCP aufbauen, vorgestellt werden.

SSL/TLS

Bei Secure Socket Layer bzw. Transport Layer Security (SSL/TLS, [RFC2246]) handelt es sich nicht um Anwendungsprotokolle im eigentlichen Sinn, sondern um Standards zur transparenten Verschlüsselung von jeglichen Anwendungsprotokollen, wobei TLS eine Bezeichnung für die neueren Varianten von SSL darstellt. In der Internetschichtenarchitektur sitzt SSL/TLS zwischen TCP und den Anwendungsprotokollen, so dass prinzipiell jede beliebige Anwendung, die keine eigenen Sicherheitsmechanismen besitzt, vollständig verschlüsselt übertragen werden kann. Auf die verwendeten kryptographischen Algorithmen soll an dieser Stelle jedoch nicht weiter eingegangen werden. SSL/TLS wird im Internet überwiegend dann verwendet, wenn das WWW-Protokoll HTTP und die Emailprotokolle POP3 bzw. IMAP gesichert eingesetzt werden sollen.

HTTP

Das Hypertext Transfer Protocol (HTTP, [RFC2616]) stellt das Standardprotokoll im World Wide Web (WWW, [TavS03]) dar und dient dort der Übertragung beliebiger Dokumente vom (Web-)Server zum (Web-)Client. HTTP arbeitet nach einem einfachen, zustandslosen Anfrage-Antwort-Schema: Eine einzelne Anfrage wird direkt beantwortet, weitere Anfragen und deren Antworten werden unabhängig davon bearbeitet. Die eindeutige Adressierung von Dokumenten im WWW wird durch einen sogenannten Uniform Ressource Locator (URL) [RFC3986] realisiert, durch den man das verwendete Zu-

griffsprotokoll, den Rechner auf dem das gewünschte Dokument liegt, den Pfad des Dokumentes auf dem Rechner und ggf. weitere Anfrageparameter angibt. In allgemeiner Form setzt sich ein HTTP-URL wie folgt zusammen:

```
http://[User[:Password]@]<Server>[:<TCP-Port>][/<Doc-Path>[?<Query>]]
```

wobei Angaben in eckigen Klammern optional sind (fehlt der TCP-Port, wird der HTTP-Standardport 80 benutzt). Ein URL lässt sich über den `<Query>`-Teil wie folgt parametrisieren:

```
<Field>=<Value>[&<Field>=<Value>]...
```

Man kann einen URL also mit beliebigen Parameter-Wert-Paaren parametrisieren, es liegt aber am Server, diese auszuwerten und die Antwort entsprechend zu gestalten. Eine HTTP-Anfrage eines Webbrowsers besteht aus einem Protokollkopf und einer Leerzeile. Der Protokollkopf enthält in der ersten Zeile die Angabe der gewählten HTTP-Methode, die Angabe der gewünschten Ressource und einer Versionsangabe, gefolgt von HTTP-spezifischen Parameterangaben, wie z. B. nochmals den Servernamen oder Angaben über die übertragene Datenmenge:

```
<Methode> <Ressource> HTTP/1.1
<Parameter>: <Wert>
```

Die gebräuchlichsten der HTTP-Methoden sind GET und POST. Bei GET müssen die URL-Parameter an eben diese angehängt werden, verwendet man POST können im Datenteil der Anfrage (nach der Leerzeile) noch weitere Parameter folgen. Das Anhängen von Binärdateien ist bei POST-Anfragen ebenfalls möglich. Falls der Webclient nur Meta-Informationen über eine Seite abfragen möchte, wird die HEAD-Methode verwendet. Mit den Methoden PUT und DELETE können Dateien auf den Server geladen oder gelöscht werden.

Eine typische GET-Anfrage lautet etwa so:

```
GET default.aspx?foo=bar&param=wert HTTP/1.1
Host: www.example.org
```

Die gleiche Anfrage mit POST formuliert lautet wie folgt:

```
POST default.aspx HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 18

foo=bar&param=wert
```

Bei der POST-Anfrage werden zusätzlich die Länge der angehängten Daten sowie deren Datenformat angegeben. Der wesentliche Vorteil von POST gegenüber GET ist die über-

tragbare Datenmenge: Während eine GET-Anfrage nur über die URL parametrisiert werden kann, deren Länge bei manchen älteren Browsern auf 256 Zeichen limitiert war, ist die Größe des POST-Datenteils nur durch die Serverkonfiguration eingeschränkt, es können Hunderte von Megabytes pro Anfrage übertragen werden.

Der Server antwortet mit einigen Statusinformationen im Protokollkopf, gefolgt von einer Leerzeile und den eigentlichen Daten. Wäre die im obigen Beispiel angefragte Ressource eine 800 Bytes große Webseite, würde die Antwort in etwa so aussehen:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=iso-8859-5
Content-Length: 800

<html><head>...</html>
```

Typischerweise enthält die Antwort noch Datums- und Sprachangaben. Weitere Details zu den einzelnen HTTP-Protokollparametern und den unterstützten Authentifizierungsmechanismen würden den Rahmen dieser Arbeit jedoch sprengen und können der offiziellen Spezifikation [RFC2616] und [RFC2617] entnommen werden.

Für den Einsatz von HTTP auch außerhalb von WWW-Anwendungen in einem Jobverteilungssystem spricht vor allem seine Flexibilität. Die Möglichkeit, beliebige Daten beliebig parametrisierbar in beide Richtungen zu übertragen, sowie die Verfügbarkeit vieler Webserver-Frameworks (etwa Java Servlets, PHP oder ASP.NET) [Str04] ermöglicht die Erstellung vielfältiger verteilter Client/Server-Anwendungen.

FTP

Das klassische File Transfer Protocol (FTP, [RFC959]) dient im Internet dazu, Dateien von einem Server zum Client oder von einem Client zum Server zu übertragen. Dabei können auch Verzeichnisse angelegt, gelöscht oder umbenannt werden. FTP wurde einzig für den effizienten Dateitransfer entworfen und ist daher eine sehr leistungsfähige Methode, im Internet beliebige Dateien mit beliebiger Größe zu übertragen. Diverse FTP-Clients existieren für alle gängigen Betriebssysteme, ebenso Serversoftware, mit denen man einen FTP-Dienst aufsetzen kann. FTP verwendet immer zwei getrennte TCP-Verbindungen: eine für die Übermittlung von Benutzerauthentifizierungs- und weiteren Steuerdaten über den (serverseitigen) Standard-TCP-Port 21, und eine für die eigentliche Nutzdatenübertragung. Dabei kommen wiederum zwei Modi zum Einsatz:

- Beim aktiven FTP wird die Steuerverbindung vom Client zum Server aufgebaut, die Datenverbindung baut jedoch der Server über einen clientseitig zufälligen TCP-Port zum Client auf, wobei der Client dem Server die Portnummer über die Steuerverbindung vorab mitteilt. Diese Variante funktioniert nicht, wenn sich der Client hinter einem NAT-Router befindet.
- Bei passivem FTP baut der Client beide Verbindungen zum Server auf, wobei für die Datenverbindung serverseitig ein zufälliger TCP-Port verwendet wird. Diese Portnummer muss vorab über die Steuerverbindung vom Server an den Client

übermittelt werden. Passives FTP funktioniert also nicht, wenn sich der Server hinter einem NAT-Router befindet.

Problematisch wird der Einsatz von FTP, wenn sich Client und Server hinter einem NAT-Router verstecken, da dann sowohl der aktive als auch der passive Modus am Nichtzustandekommen der Datenverbindung scheitern. FTP arbeitet wie HTTP vollständig unverschlüsselt, kann jedoch ebenso über eine SSL/TLS-Verbindung gesichert werden. Die Flexibilität von HTTP, Daten beliebig mit Metadaten parametrisierbar zu übertragen, besitzt FTP nicht.

SSH

Die Secure Shell (SSH) [RFC4251] dient dem textbasierten Remote-Login auf entfernte Rechnersysteme und kommt überwiegend in der Linux-/Unixwelt zum Einsatz. SSH sichert die gesamte Kommunikation selbst und ist damit nicht auf Sicherheitsmechanismen in tieferen Netzwerkschichten angewiesen. Neben der Remote-Shell bietet SSH auch die Möglichkeit, beliebige TCP-Verbindungen zu tunneln, Prozesse auf dem Remote-System zu starten oder gesicherte Dateitransfers (SSH Secure File Transfer, SFTP) anzustoßen. Über Public-Key Authentifizierung ist es möglich, sich automatisiert und ohne Angabe von Benutzerpasswörtern auf einem entfernten System remote einzuloggen.

Dies und die oben erwähnte Option, auf dem Remotesystem Prozesse zu starten, machen SSH für den Einsatz in einem Jobverteilungssystem interessant. Die Tatsache, dass dazu allerdings auf jedem Rechenknoten ein SSH-Serverdienst eingerichtet werden müsste (was in der Windows-Welt unüblich ist) und die Tatsache, dass man auf dem Remote-System einen vollen interaktiven Shell-Zugang hätte und der Dienst hinter einem NAT-Router nicht erreichbar wäre, sprechen jedoch dagegen.

SMB/CIFS

Server Message Block (SMB) bezeichnet ein Protokoll für Dateidienste in lokalen Netzen. Es wurde 1983 von IBM vorgestellt und ist heute das Standardprotokoll zum Remote-Zugriff auf Dateien und Druckern in Windows Netzwerken. Die 1996 von Microsoft vorgestellte Erweiterung Common Internet File System (CIFS) [CIFS02] bietet neben der ursprünglichen Funktion als verteiltes Dateisystem weitere Dienste wie Remote Procedure Calls (siehe 2.2.2), Inter-Prozess-Kommunikation und den (heute durch Active Directory ersetzten) Domänendienst von Windows NT 4. CIFS setzt im Gegensatz zum ursprünglichen SMB direkt auf TCP/IP auf, es existieren außerhalb von Windows viele Implementierungen für entsprechende Clients und Serverdienste, die Bekannteste davon ist Samba.

2.2.2 Middleware für verteilte Anwendungen

Ein System zum Ausführen rechenintensiver Jobs auf entfernten Knoten lässt sich als verteiltes System auffassen, bei dem Daten (Dateien, Befehle etc.) vom PC des Nutzers auf viele Rechner und wieder zurück zum Nutzer übertragen werden müssen. Neben den oben vorgestellten Standardprotokollen im Internet existieren verschiedene Techniken, mit denen sich eigene, auf die Aufgabe maßgeschneiderte verteilte Systeme realisieren

lassen. Solche Middleware abstrahiert von der eigentlichen Übertragungstechnik, erspart dem Entwickler den Entwurf und die Implementierung eines Anwendungsprotokolls und bietet höhere Funktionen, wie man sie von gängigen Programmiersprachen her kennt. Ein Beispiel wäre ein Methodenaufruf mit Objektzeigerparametern. Im Folgenden sollen einige solcher Techniken vorgestellt werden, wobei die Abschnitte über Remote Procedure Calls (RPC) und Java Remote Method Invocation (RMI) an [TavS03] und der über .NET Remoting an [HuSc04] angelehnt sind.

Remote Procedure Calls

Unter einem Remote Procedure Call innerhalb eines verteilten Systems versteht man einen Funktionsaufruf, bei dem die einzelnen Parameter beim Aufrufer von einer speziellen RPC-Bibliothek in eine Netzwerknachricht verpackt und zum aufgerufenen Rechner übertragen werden. Dort werden die Parameter wiederum von einer speziellen RPC-Bibliothek aus der Nachricht extrahiert und an eine lokale Funktion übergeben, wo dann die eigentliche Prozedur physisch ausgeführt wird. Am Ende der Ausführung durchläuft das Funktionsergebnis die gleichen Schritte zum Aufrufer zurück. Für einen Programmierer sind RPCs sehr praktisch: Netzwerkkommunikation vereinfacht sich so zu Funktionsaufrufen einer lokalen Bibliothek (dem sogenannten Stub), welche die Komplexität der Netzwerkkommunikation vollkommen vor dem Aufrufer versteckt. RPCs sind in verteilten Systemen weit verbreitet, z. B. arbeitet das Network File System NFS der Firma Sun RPC-basiert. Die Fernadministration von Windows NT und seinen Nachfolgern geschieht ebenfalls überwiegend auf Basis von RPCs.

Problematisch bei RPCs ist die Tatsache, dass die Datentypen so über das Netzwerk übertragen werden, wie sie im Rechner vorliegen: Unterschiede zwischen Programmiersprachen, Betriebssystemen und Prozessorarchitekturen führen zu fehlinterpretierten Parametern. Beispielsweise kodieren moderne Sprachen wie C# oder Java Zeichenketten in 16-Bit Unicode, während C/C++ Zeichenketten als Array aus 8-Bit Zeichen darstellen. Teilweise Abhilfe schaffen RPC-Standards wie z. B. DCE (Distributed Computing Environment), bei denen zunächst die Schnittstelle eines RPC-Dienstes in einer sogenannten Interface Definition Language (IDL) spezifiziert wird, aus der anschließend ein spezieller Compiler die Client- und Serverstubs generiert. Diese Stubs übersetzen die lokale Datendarstellung in ein standardisiertes Übertragungsformat.

Java Remote Method Invocation

Der grundsätzliche Unterschied von Java RMI zum traditionellen RPC ist die Objektorientiertheit: Entfernte Objekte können rechnerübergreifende Objektreferenzen unterstützen. Diese Referenzen werden als Methodenparameter übergeben. Ähnlich zu RPCs steht bei RMI clientseitig ein Stellvertreterobjekt (Proxy) zur Verfügung, das wie ein lokales Objekt benutzt werden kann und das die eigentlichen Methodenaufrufe an den Server und damit an das dortige entfernte Objekt überträgt. Bei den entfernten Methodenaufrufen können nicht nur einfache Datentypen wie Zahlen oder Zeichenketten übergeben werden, sondern komplexe Objekte mit eigener interner Funktionalität, sofern sie serialisierbar sind. Unter Serialisierung versteht man das Verpacken eines Objektes mitsamt

seinen internen Zuständen und Methoden in ein (Binär-)Format, das über ein Netzwerk übertragen werden kann. Bei Java RMI sind sogar die Proxies serialisierbar, so dass diese nicht schon während der Entwicklung beim Client vorhanden sein müssen. Dabei ist man jedoch auf die Verwendung der Java-Sprache und der Java-Plattform angewiesen, sowohl beim Client als auch beim Server.

.NET Remoting

Verwendet man bei der Implementierung eines verteilten Systems das Microsoft .NET Framework [Sch06a, Tro02], lässt sich Prozess- und Netz-übergreifende Kommunikation mit dem dort integrierten Remoting Mechanismus realisieren. Dessen Art der Nutzung verhält sich ähnlich wie sein Pendant Java RMI: Es lassen sich damit verteilte objektorientierte Anwendungen vergleichsweise einfach entwickeln. Vollkommen transparent werden Objekte, Referenzen auf Objekte, Methodenaufrufe mitsamt ihren Parametern und deren Rückgabewerte über ein TCP-Netz an einen entfernten Rechner übertragen bzw. dort ausgeführt. Unterschieden wird hierbei nach dem Ort der Instanziierung des Remote-Objekts: Serialisierbare Objekte können clientseitig erzeugt und dann zum Server zur entfernten Ausführung übertragen werden, oder aber der Server stellt Remote-Objekte bereit. Werden die Objekte vom Client erzeugt, bleiben sie exakt diesem zugeordnet, weswegen in diesem Fall die Implementierung der Klasse beim Client vorhanden sein muss. Werden die Objekte serverseitig erzeugt, reicht dem Client eine Schnittstellendefinition, aus der ein Proxyobjekt generiert wird, das die Methodenaufrufe zum Server weiterleitet, wo sie dann ausgeführt werden. Die Verwendung von .NET-Remoting legt den Entwickler zwar nicht auf eine Sprache fest, jedoch auf das .NET-Framework, das trotz alternativer Implementierungen noch nicht die Betriebssystemunabhängigkeit von Java erreicht hat.

Webservices

Unter einem Webservice versteht man einen webbasierten Dienst, der den Zweck hat, automatisiert Daten zwischen Anwendungen über ein Netzwerk auszutauschen [DJMZ05, KuWö02]. Sie kommen dabei primär in Client/Server-Architekturen zum Einsatz, wobei keine explizit für Menschen erstellte Daten übertragen werden (z. B. HTML-Seiten im WWW), sondern maschinengeneriertes XML (Extensible Markup Language) und lassen sich deshalb gut zur Anwendungsintegration in heterogenen Plattformen nutzen. Alle verwendeten Datenmodelle und Protokolle basieren ausschließlich auf XML und sind allgemein standardisiert. Webservices werden eingesetzt, um auf einem entfernten Rechner einen Dienst zu nutzen, wobei sowohl die Dienstschnittstelle als auch die eigentlichen Nutzdaten in XML definiert bzw. codiert werden. Einen Webservice kann man sich gut als entfernten Methodenaufruf vorstellen, bei dem die Methodenparameter zur Übertragung in XML serialisiert werden. Die drei grundlegenden Standards für Webservices sind UDDI, WSDL und SOAP, welche im Folgenden kurz beschrieben werden.

UDDI: Beim Universal Description, Discovery and Integration (UDDI) Standard handelt es sich selbst um einen (Meta-)Webservice, der als Verzeichnis für im Internet angebotene

Webservices dient [UDDI04]. Eine wichtige Funktion ist die Möglichkeit, nach Diensten zu suchen, die bestimmte technische oder inhaltliche Anforderungen erfüllen sollen.

WSDL: Mit der ebenfalls XML-basierten Web Service Description Language (WSDL) werden Webservices maschinenlesbar beschrieben. In einem WSDL-Dokument wird genau spezifiziert, welche Methoden ein Dienst bereitstellt, welche Parameter er erwartet, wie diese in XML abgebildet werden müssen und welche Daten an den Aufrufer zurückgeliefert werden [WSDL01]. Hierbei sind eigene XML-Schemadefinitionen möglich, mit denen man exakt angeben kann, auf welche Art komplexe Datentypen wie etwa Arrays in XML zu kodieren sind. Weiterhin wird in der Schnittstellenbeschreibung angegeben, wie der Dienst technisch anzusprechen ist, z. B. das benötigte Datenformat, die Adresse und das Anwendungsprotokoll, über die der Webservice und seine Methoden erreichbar sind, üblicherweise eine oder mehrere URLs. Die WSDL-Datei eines Webservice kann clientseitig dazu genutzt werden, programmiersprachenspezifischen Proxycode zu erzeugen, der lokale Funktionsaufrufe in SOAP übersetzt und an den entfernten Dienst überträgt. Aus der Sicht eines Entwicklers lassen sich Webservices damit sehr leicht ansprechen.

SOAP: Der früher Simple Object Access Protocol genannte Standard (heute ist SOAP ein alleinstehender Begriff) dient dem Austausch von beliebigen strukturierten Daten zwischen Anwendungen über ein Netzwerk [SOAP07]. Die Daten werden dabei vollständig in XML dargestellt, so dass sich in eine einzelne Nachricht auch sehr komplexe und umfangreiche Daten einbinden lassen. Die Syntax von SOAP-Nachrichten wird von W3C-Standards und den WSDL-Beschreibungen eines Dienstes festgelegt. Wie WSDL auch beschreibt SOAP nur die technischen Details einer Nachricht, Semantikinformation fehlt. SOAP definiert auch kein Übertragungsverfahren, weswegen die Nachrichten prinzipiell über jedes Internetprotokoll übertragen werden können, wobei in der Praxis meist HTTP verwendet wird, das die SOAP-Nachrichten im Datenteil einer POST-Anfrage überträgt. SOAP kann als Nachfolger des (weniger flexiblen, aber dafür deutlich einfacher gehaltenen) XML-RPC Standards angesehen werden, bei dem die Nutzdaten eines entfernten Prozeduraufrufs ebenfalls im XML-Format übertragen werden.

Aufgrund der breiten Standardisierung der Webserviceprotokolle und der ausschließlichen Verwendung von XML eignen sich Webservices vor allem zum Einsatz in heterogenen Betriebssystem-, Programmiersprachen- und Netzwerkumgebungen, einschließlich des Internets. XML-Parser existieren für viele Plattformen und durch das zum Transport meist verwendete Web-Protokoll HTTP stellen Firewalls und NAT-Router üblicherweise kein Problem dar. Insbesondere lassen sich die HTTP-eigenen Funktionen wie Verschlüsselung per SSL/TLS oder Authentifizierung ebenfalls nutzen, auch wenn hierfür inzwischen spezielle auf SOAP basierende Mechanismen (wie z. B. WS-Security) standardisiert wurden. Ein Nachteil von Webservices (insbesondere von SOAP) ist der potentiell große Overhead bei der Datenbeschreibung und die Tatsache, dass beide Kommunikationspartner XML-Code parsen müssen, was rechenintensiv sein kann. Durch die Möglichkeit, per XML-Schema beliebig komplexe Datentypen in der WSDL-Datei eines Dienstes spezifizieren zu können, lassen sich jedoch aufwändige Transaktionen in einer einzelnen SOAP-Nachricht übermitteln, so dass der Overhead weniger ins Gewicht fällt.

2.2.3 Parallele Programmierung

Neben den bisher vorgestellten Methoden zur Entwicklung prinzipiell beliebiger verteilter Systeme sollen nun zwei Techniken vorgestellt werden, die explizit dafür entworfen wurden, rechen- und datenintensive parallele Anwendungen zu entwickeln. Dabei liegt der Fokus nicht auf Verfahren, die ein Programm für (symmetrische) Mehrprozessorsysteme parallelisieren (wie z. B. Multithreading-Bibliotheken), sondern auf Methoden, die den prozessübergreifenden Datenaustausch vereinheitlichen und die die zu einer Anwendung gehörenden Tasks gleichzeitig auf verschiedenen Rechensystemen im Netzwerk ausführen. Am weitesten verbreitet sind das Message Passing Interface (MPI) und die Parallel Virtual Machine (PVM), auf die im Folgenden eingegangen wird.

MPI

Das Message Passing Interface (MPI) [MPI07] beschreibt keine explizite Software, sondern einen Schnittstellenstandard zur nachrichtenorientieren Interprozesskommunikation, für den verschiedenste Implementierungen existieren. Dies betrifft sowohl unterschiedliche Betriebssysteme als auch Programmiersprachen, mit denen man parallele Applikationen entwickeln kann. Weiterhin existieren auf eine bestimmte Hardware zugeschnittene Realisierungen mit dem Ziel, die Kommunikationsperformanz zu erhöhen. MPI-Anwendungen werden üblicherweise auf Großrechnern eingesetzt, wo der gleiche Prozess mehrfach parallel auf den jeweiligen Rechenknoten gestartet wird. Diese Prozesse kommunizieren dann untereinander durch Austausch von Nachrichten, wobei über spezielle Hardwaretreiber der direkte Zugriff auf den Hauptspeicher des Zielsystems möglich ist. Die Kommunikation per TCP ist ebenfalls möglich. Der neuere Standard MPI-2 ermöglicht es unter anderem, zur Laufzeit dynamisch weitere Prozesse auf Remotesystemen zu erzeugen sowie Gruppenkommunikation. Die Zuordnung von Prozessen zu Knoten erfolgt zu Beginn der parallelen Ausführung durch den Prozessmanager der Laufzeitumgebung, der die einzelnen Prozesse verteilt und die Topologie festlegt: Jeder Prozess muss über Informationen verfügen, auf welchem Prozessor er läuft und welche Nachbarprozesse auf welchen Nachbarprozessoren existieren und wie diese adressierbar sind. Damit das funktioniert, muss auf jedem Rechenknoten ein MPI-Dienst gestartet werden, die einzelnen Dienstprogramme und deren TCP/IP-Adressen werden über spezielle Befehle der Laufzeitumgebung vorab miteinander bekannt gemacht [Lau06].

Aufgrund der hohen Skalierbarkeit und der Portabilität sind MPI-synchronisierte verteilte Prozesse die übliche Lösung für auf Hochleistungsrechnern laufende parallele Anwendungen. Die Implementierungen sind jedoch sprach- und oft auch betriebssystemspezifisch, wobei es überwiegend Implementierungen für C/C++ und Fortran gibt. Weiterhin abstrahieren sie meist nicht genug von der verwendeten Hardware, so dass Interoperabilität oft nicht gegeben ist. Für den Einsatz auf Höchstleistungsrechnern spielt das natürlich keine Rolle, in heterogenen Umgebungen lassen sich MPI-Anwendungen deswegen jedoch nur schwer realisieren. Da der MPI-Standard keinerlei objektorientierte Merkmale besitzt, gestalten sich Portierungen auf z. B. Java oder .NET schwierig. MPI-2 schafft hier mit der Spezifikation sprachunabhängiger Objekte ein wenig Abhilfe.

PVM

Bei der Parallel Virtual Machine (PVM) [GBDW+94] handelt es sich um ein Software-Framework, das es ermöglicht, parallele Anwendungen auf heterogenen Rechnersystemen zu verteilen. PVM ist kein implementierungsloser Schnittstellenstandard wie MPI, sondern ein konkretes Produkt und besteht im Wesentlichen aus einem Hintergrunddienst, der auf jedem Rechenknoten installiert werden muss, einer Bibliothek zur Anwendungsentwicklung und einigen Kommandozeilentools zum Verwalten der parallelen Anwendung(en). PVM arbeitet prozessorientiert: Jede parallele Anwendung besteht aus einer Menge serieller Tasks, die mit Hilfe der Routinen der PVM-Bibliothek Nachrichten austauschen. Dem Entwickler einer solchen Anwendung stehen dabei viele Möglichkeiten offen: Neben dem Bearbeitungsmodell, bei dem eine Menge an voneinander unabhängigen Jobs auf den gleichen Daten die gleichen, aber unterschiedlich parametrisierten Operationen ausführen, sind auch komplexe Parallelisierungstechniken möglich, wie beispielsweise das baumartige Aufteilen einer Anwendung, wobei die Baumtiefe zur Entwicklungszeit noch nicht bekannt sein muss. Analog zu MPI erfolgt die Parallelisierung bereits bei der Anwendungsentwicklung, der Entwickler muss also vorab wissen, welche Codeteile auf einen anderen Rechner ausgelagert werden können und wie die parallelen Teile synchronisiert werden müssen.

Die nachrichtenorientierte parallele Bearbeitung von Tasks funktioniert wie bei MPI natürlich nur, wenn auf jedem Rechenknoten bekannt ist, welche anderen Knoten zur Verfügung stehen. Ebenso ist ein gemeinsames Dateisystem notwendig, über das die kompilierten Programme auf die Zielplattform übertragen werden. PVM ist sehr unixlastig, es existieren allerdings auch Portierungen für Windows. Die Programmierbibliotheken standen ursprünglich nur für C und Fortran zur Verfügung, inzwischen existieren auch Portierungen auf andere Sprachen, wie z. B. Perl, Python oder Java.

2.3 Jobverteilung bei Höchstleistungsrechnern und Rechenclustern

Die in diesem Abschnitt beschriebenen Rechensysteme lassen sich vor allem dadurch charakterisieren, dass sie nur dem einen Zweck dienen: Maximale Rechenleistung zur Durchführung wissenschaftlicher Berechnung bereitzustellen. Handelt es sich dabei um einen Höchstleistungsrechner (umgangssprachlich auch Supercomputer genannt), werden meist die zum Installationszeitpunkt leistungsfähigsten verfügbaren Hardwarekomponenten eingesetzt. Dies betrifft zwar auch die Prozessoren, von denen oft hunderte oder tausende zum Einsatz kommen, aber vor allem die Netzwerkstrukturen, mit denen die einzelnen Rechenknoten verbunden werden: Spezielle sogenannte Interconnects wie QSNet [Qsn04], Infiniband [Inf07] oder Myrinet [Myr07] übertreffen die üblichen für lokale Netze verwendeten Techniken (TCP/IP, Gigabit-Ethernet) hinsichtlich Übertragungsgeschwindigkeit und Latenzzeit deutlich [BHB04]. Demgegenüber bestehen dedizierte Rechencluster oft aus Standard-PCs, die mit Internettechnik (also über Ethernet übertragenes TCP/IP) miteinander gekoppelt sind. Beiden gemeinsam ist die Homogenität der eingesetzten Hard- und vor allem der Software: Betriebssystem, Compiler, Bibliotheken (funktionale wie MPI oder numerische wie z. B. Atlas [WPD01]) und das Verteil-/Batchsystem sind auf allen Rechenknoten identisch.

Die auf diesen Rechnern eingesetzten Verteilsysteme haben die schwierige Aufgabe zu lösen, die Berechnungsaufträge der um die Ressourcen konkurrierenden Benutzer des Systems auf die Rechenknoten zu verteilen. Dabei müssen unter anderem vom Benutzer vorgegebene Deadlines, Priorisierungen oder der erforderliche Parallelitätsgrad bei MPI-Anwendungen berücksichtigt werden. Die Leistung von Höchstleistungsrechnern wird in manchmal auch weiterverkauft. In dem Fall muss das Verteilsystem beachten, dass ein Benutzer auch nur so viele Rechenknoten oder Rechenzeit erhält, wie er bezahlt hat. Es existiert eine Menge Literatur, die sich mit der Optimierung dieser Problemstellungen befasst [CPPM+07, FPFP06, GRV04, HHL07, MASH+99, KEF01, TEF07], meist setzen diese Optimierungen auf den typischen, in Großrechnern und Clustern eingesetzten Verteilsystemen wie PBSpro, MAUI oder die Sun Grid Engine auf [EtTs05].

Diesen Batchsystemen und ihren Scheduling-Algorithmen ist gemeinsam, dass sie sich auf gewisse Randbedingungen verlassen oder verlassen können:

- Alle verfügbaren Rechenknoten sind bekannt und deren Anzahl ändert sich nicht.
- Die Leistungsfähigkeit jedes Rechenknotens ist bekannt, steht ausschließlich für die Rechenjobs zur Verfügung und ist zwischen zwei Knoten vergleichbar. Das bedeutet insbesondere, dass man vorhersagen kann, wie lange ein Job auf einem Knoten B benötigt, wenn seine Laufzeit auf Knoten A bekannt ist.
- Das Verteilsystem kann auf jedem Knoten direkt Jobs starten, z. B. durch eine Remote-Login-Möglichkeit oder einen speziellen Dienst, der auf jedem Knoten läuft und Jobs entgegennimmt.
- Ein Rechenknoten, auf dem ein Job erfolgreich gestartet werden kann, führt diesen zuverlässig bis zum Ende aus, ohne während der Berechnung auszufallen.

Mit Ganglia [MCC03] und Nagios [Bar05] existieren Monitoring-Systeme, die eingesetzt werden, um die Funktionsfähigkeit und Auslastung der einzelnen Knoten eines Clusters zu protokollieren. Die verschiedenen erfassten Parameter werden jedoch aus naheliegenden Gründen nicht beim Scheduling verwendet: Die Rechner arbeiten alle zuverlässig. Es gibt jedoch auch Rechner oder Rechnerverbände, in denen solche Voraussetzungen nicht unbedingt gegeben sind. Dies ist Gegenstand des nächsten Abschnitts.

2.4 Systeme zur Jobverteilung auf PCs

Der Schwerpunkt der vorliegenden Arbeit liegt nicht auf den im vorhergehenden Kapitel beschriebenen Verteilsystemen für Höchstleistungsrechner oder speziell dafür eingerichteten Rechenclustern, sondern auf Systemen, die es ermöglichen, brachliegende Rechenleistung auf Rechnern zu nutzen, die nicht (oder zumindest nicht ausschließlich) für wissenschaftliche Berechnungen bestimmt sind. Sind nur gelegentlich Jobs durchzurechnen oder spielt kürzeste Berechnungszeit bei maximaler Performance nur eine untergeordnete Rolle, bietet es sich an, normale Arbeitsplatzrechner oder Server, die wenig ausgelastet sind, dafür zu nutzen. Bei diesem sogenannten *High Throughput Computing* kommt es im Gegensatz zum *High Performance Computing* nicht darauf an, maximale Computerleistung in allen Bereichen (CPU, Speicher, Netzwerkgeschwindigkeit) kurzfristig für eine

einzelne Aufgabe zur Verfügung zu haben, sondern darauf, eine große Anzahl von Berechnungen durchzuführen, und zwar unter der Voraussetzung, dass man viel Zeit hat (und aufgrund der Anzahl der Berechnungen auch braucht): Es stellt sich nicht die Frage, wie viele Teraflops für einige Tage zur Verfügung stehen, sondern wie viele Jobs man in den nächsten drei Monaten berechnen kann.

Die Hard- und Softwarevoraussetzungen sind dabei andere als bei Höchstleistungsrechnern und dedizierten Rechenclustern:

- Die Prozessoren der beteiligten Rechner unterscheiden sich in der Leistungsfähigkeit teilweise signifikant, oft bis zu einem Faktor von 5–10.
- Die Architektur der Prozessoren kann unterschiedlich sein. In der Regel kommen x86-CPUs zum Einsatz, jedoch können sich bereits diese beträchtlich unterscheiden: Cachegröße, Anzahl Kerne, Adressierungsbreite (32/64 Bit) sind verschieden.
- Bei den Netzwerkverbindungen handelt es sich praktisch immer um ein TCP/IP-Netz, das hardwareseitig über Fast- oder Gigabit-Ethernet realisiert ist.
- Die auf den einzelnen Rechenknoten installierten Betriebssysteme und sonstigen Softwarekomponenten sind nicht homogen.
- Die beteiligten Rechner erledigen oft noch andere Aufgaben, als lediglich möglichst viele Jobs zu berechnen.

Im Folgenden werden drei ausgewählte Systeme beschrieben, die sich zum Ziel gesetzt haben, brachliegende Rechenleistung im Arbeitsplatzumfeld zu nutzen: *Condor*, weil es sich durch die breite Unterstützung vieler Hardwareplattformen und Betriebssystemen vor allem beim Einsatz in heterogenen Systemen gut eignet und dabei ein breites Feld von Programmier- und Skriptsprachen unterstützt; *BOINC*, weil es durch seine Netzwerkarchitektur uneingeschränkt im Internet einsetzbar ist und sich als populäre „Public-Ressource-Computing[2]"-Plattform weit verbreitet hat und *Alchemi*, weil damit die Entwicklung und Verteilung einer parallelen Anwendung mit modernen, objektorientierten Konzepten möglich ist. Mit *Entropia* [CCEB03] und *XtremWeb* [GNFC00] existieren weitere vergleichbare Systeme, die jedoch im Rahmen dieser Arbeit nicht weiter betrachtet werden.

2.4.1 Condor

Condor [LLM88, TWML01] wird seit 1988 an der University of Winsconsin Madison entwickelt. Ursprünglich im Rahmen des Remote-Unix-Projektes dazu gedacht, verteilte Ressourcen auf Unix Arbeitsstationen zu nutzen, wurde es mit dem Ziel, brachliegende CPU-Zeit auf beliebigen Arbeitsplatzrechnern für lang laufende Jobs verfügbar zu machen, zu einem flexiblen Lastverteilungssystem weiter entwickelt. Sowohl reine Batchjobs als auch echte parallele Anwendungen, die PVM oder MPI nutzen, werden unterstützt. Condor besitzt einen flexiblen und mächtigen Mechanismus, mit dem man detailliert angeben kann, welche Systemvoraussetzungen ein Job benötigt, um laufen zu können

[2] Ein Verteilungsmodell, bei dem jeder beliebige Internet-PC an der Bearbeitung massiv parallelisierbarer, sehr rechenintensiver Probleme teilnimmt.

bzw. welche Arten von Jobs ein Rechenknoten ausführen kann. Transparente Dateisystemzugriffe sowie transparentes Checkpointing und Migration von Jobs auf andere Rechner sind ebenfalls vorhanden. Zunächst seien jedoch die grundsätzliche Architektur und die Arbeitsweise eines Condor Rechnerverbundes (Condor-Pool) beschrieben.

Architektur und Arbeitsweise

Die Rechner einer Condor-Installation (Abb. 2.5) übernehmen eine oder mehrere Aufgaben, welche im Folgenden genauer beschrieben werden[3].

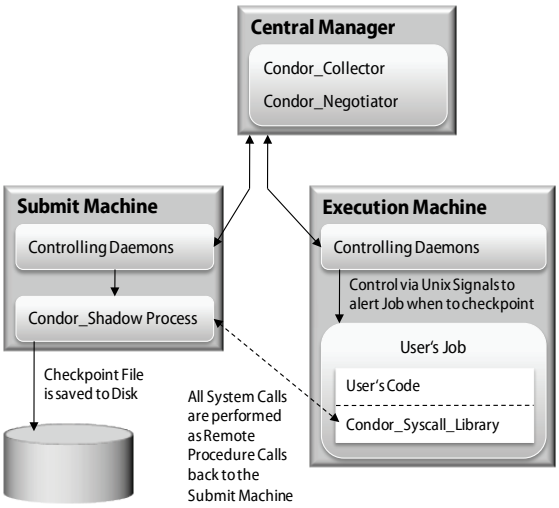


Abb. 2.5 Condor Architektur[4]

Central Manager: Der Manager, der in jedem Condor-Pool nur auf einem einzigen Rechner installiert werden muss, weiß jederzeit darüber Bescheid, welche Rechner im Pool Jobs ausführen können, und welche Arten von Jobs das jeweils sind. Weiterhin sind hier die Beschreibungen aller auszuführenden Jobs hinterlegt. Die Aufgabe des Central Manager besteht darin, zu entscheiden, welcher Job auf welchem Rechner ausgeführt wird. Dabei wird kein klassisches Scheduling betrieben: Man versucht nicht etwa, Leerlaufzeiten zu minimieren oder Jobdeadlines einzuhalten, sondern es geht vielmehr darum, die Anforderungen eines Jobs und das Rechnerangebot möglichst gut miteinander abzustimmen. Fällt der Central Manager aus, können keine neuen Jobs mehr verteilt werden, Laufende werden jedoch zu Ende ausgeführt.

Submit Machine: Diese Rechner sind sozusagen der Zugangspunkt für die Nutzer, denn hier werden die einzelnen Jobs an das System übergeben. Jede Submit Machine verwaltet eine Warteschlange der ihr übergebenen Jobs und führt diese dann auf den Rechnern aus, die ihr vom Central Manager zugewiesen werden. Für jeden remote ausgeführten Job läuft auf einer Submit Machine ein Schattenprozess. Da von hier aus viele Jobs gleichzeitig an die ausführenden Rechner verteilt und die Datei Ein- und Ausgaben eines jeden laufenden Jobs entgegengenommen werden, braucht ein solcher Rechner sehr viel

[3] Auf die Eindeutschung der Rollenbezeichnungen wurde dabei verzichtet.
[4] Quelle: Nach Condor Handbuch auf [Con07]

Hauptspeicher und Festplattenplatz. Der Schattenprozess auf der Submit Machine läuft dabei immer so lange, wie der eigentliche Job auf dem ausführenden System läuft. Jeder Rechner, von dem aus Jobs an Condor übergeben werden, muss also für dessen kompletten Ausführungszeitraum zu Verfügung stehen.

Execution Machine: Die Rechner, die diese Rolle einnehmen, führen letztlich die einzelnen Jobs aus. Jeder Rechner im Pool, auch der Central Manager oder die Submit Machines können zusätzlich zu ihren primären Aufgaben Jobs ausführen. Execute Machines senden periodisch ihre Zustandsbeschreibung an den Central Manager.

Ausführungsumgebungen für Jobs

Entsprechend den Anforderungen, die ein Nutzer an Condor stellt, muss man einen Job einer der unterschiedlichen Ausführungsumgebungen zuweisen: Condor besitzt mehrere sogenannte *Universen*, die sich bezüglich der Anforderungen an einen Job und der gebotenen Funktionalität unterscheiden. Die wichtigsten Universen und deren Fähigkeiten werden im Folgenden vorgestellt.

Standard-Universum: Jobs, die im Standardkontext ausgeführt werden können, profitieren am meisten von Condors Fähigkeiten, denn es stehen transparentes *Checkpointing* und *Remote System Calls* zu Verfügung (vgl. Abb. 2.5).

- *Checkpointing:* Condor kann einen Standardjob jederzeit einfrieren, den gesamten Zustand auf eine andere Ausführungsmaschine übertragen und den Job dort weiterlaufen lassen. Damit können Jobs – gegen Rechnerausfälle geschützt – monatelang auf wechselnden Rechnern ausgeführt werden, man muss nur dafür sorgen, dass die Submit Machine mit dem Schattenprozess immer läuft.
- *Remote System Calls:* Dateisystemoperationen auf der Ausführungsmaschine werden an die Submit Machine zurückgetunnelt, dort physikalisch ausgeführt und die Daten werden dann ans Remotesystem zurückübertragen. Dem Job wird damit quasi die Umgebung des Nutzers vorgetäuscht: Auf Nutzerdaten auf der Submission Machine kann in völlig transparenter Art zugegriffen werden, dort muss allerdings ebenfalls permanent der Schattenprozess laufen.

Die Nachteile dieser Ausführumgebung bestehen nun nicht nur darin, dass die Submit Machine ebenfalls permanent laufen muss (das ist bei allen Condor-Jobs in allen Universen so), sondern in der potentiell hohen Netzwerklast durch die RPCs und darin, dass die Fähigkeit zum transparenten Sichern des Jobzustandes und zum Tunneln von Systemaufrufen unter anderem die folgenden strengen Anforderungen an die Jobs stellt:

- Standardjobs dürfen sich nicht in mehrere Prozesse aufspalten.
- Interprozess- und Netzwerkkommunikation sind nicht möglich.
- Der Aufruf vieler verschiedener Systemfunktionen ist nicht erlaubt.
- Die Anwendung muss im Quellcode vorliegen und beim Kompilieren mit einer speziellen Condor-Bibliothek verlinkt werden, welche für C/C++ und Fortran verfügbar ist.

- Das Standard-Universum steht nur auf den Betriebssystemen HP-UX 10.20, Solaris 8/9 und diversen Red-Hat-Linux Varianten zur Verfügung.

Das Standard-Universum ist die älteste Ausführumgebung und ging direkt aus dem ursprünglichen Remote-Unix-Projekt hervor, was sich in der Unixlastigkeit der dort lauffähigen Jobs widerspiegelt. In einem heutigen Pool aus Arbeitsplatzrechnern, die im Wesentlichen mit neueren Windowsvarianten oder aktuellen Linuxdistributionen betrieben werden, und bei Anwendungen, die meist nicht ausschließlich in C geschrieben sind, würde man die Jobs wohl eher in einer der folgenden Umgebungen ausführen lassen.

Vanilla-Universum: Hat man keine Möglichkeit, die auszuführende Datei mit der Condor-Bibliothek zu verlinken, oder scheidet die Benutzung des Standard-Universums aufgrund der oben genannten Einschränken aus, nutzt man die Vanilla-Umgebung. Sie dient dem Remotestart beliebiger ausführbarer Dateien. Da der RPC-Mechanismus für Dateizugriffe deswegen nicht funktioniert, ist zum Datenaustausch entweder ein zentrales gemeinsames Dateisystem wie z. B. NFS nötig, oder aber man nutzt den Condor-eigenen Dateitransfermechanismus, der alle Quelldateien vom Zugangsrechner zum Rechenknoten und die Ergebnisdateien von dort wieder zurückkopiert. Kommt ein gemeinsames Dateisystem zum Einsatz, muss sich der von Condor gestartete Prozess am Dateisystem-Server authentifizieren können. Auch bei Vanillajobs ist es notwendig, dass auf der Submit Machine während der Ausführung permanent ein Schattenprozess läuft.

Java-Universum: Diese speziell auf die Ausführung von in Java geschriebenen Jobs ausgelegte Umgebung entspricht im wesentlichen dem Vanilla-Universum, erweitert um einige Java-spezifische Funktionen, die unter anderem das Ausführen von .jar-Dateien, das Setzen des Classpaths und die Ausnahmebehandlung betreffen.

DAGMan-Universum: Condor selbst bietet keine Funktionalität, bei der Verteilung von Jobs Abhängigkeiten zu modellieren und Jobs entsprechend dieser Abhängigkeiten zu verteilen. Um Jobpakete zu definieren, bei denen die Ausführung eines Jobs von Vorbedingungen, wie z. B. dass ein anderer Job schon Ergebnisse produziert hat, abhängt, benötigt man den Metascheduler DAGMan, der als azyklischer gerichteter Graph (directed acyclic graph, DAG) modellierte Jobketten entgegennimmt und die Jobs entsprechend den dort definierten Abhängigkeiten an Condor übergibt.

Parallel-Universum: Condor bietet Funktionen zur Unterstützung von MPI oder PVM-Jobs (vgl. 2.2.3), bei denen Interprozesskommunikation zwischen mehreren Knoten erfolgt und bei denen deswegen eine bestimmte Menge von Prozessen gleichzeitig auf verschiedenen Rechnern ausgeführt werden muss. Bei PVM-Jobs, die nach dem Master-Worker-Prinzip arbeiten, läuft beispielsweise der Masterjob auf der Submit Machine, während die Workerjobs an die Arbeitsrechner delegiert werden.

Verteilverfahren

Entscheidend für die Zuteilung von Jobs zu Ausführungsmaschinen sind bei Condor sogenannte *ClassAds*. Damit sind Beschreibungsdateien gemeint, die sehr genau spezifizieren, welche Anforderungen ein Job an einen Rechenknoten stellt, auf dem er ausgeführt wird. Umgekehrt kann man sehr genau festlegen, welche Art von Job auf einem bestimm-

ten Knoten ausgeführt werden soll. Der Besitzer eines Rechenknotens kann so sehr genau angeben, in welcher Art und Weise sein Rechner für Condor-Jobs eingesetzt werden soll. Das betrifft einerseits technische Dinge, wie maximaler Speicherverbrauch oder nutzbare CPU-Anzahl, andererseits auch Vorgaben bezüglich der Einsatzzeit oder der unterstützten Benutzergruppen. Demgegenüber muss der Nutzer bei seinen Jobs zwingend vorgeschriebene Anforderungen bezüglich der benötigten Plattform spezifizieren, kann aber auch optional Wünsche bezüglich der ausführenden Maschine angeben.

Ein ClassAd besteht immer aus einer Menge von Attribut-Wert-Paaren, wobei die Werte auch aus komplexen mathematisch-logischen Ausdrücken bestehen und auf andere Attribute verweisen dürfen. Die Angaben sind flexibel und können unter anderem auch Makros enthalten, die erst zur Laufzeit evaluiert werden. Das Beispiel einer Jobbeschreibung in Listing 2.1 soll die Arbeitsweise der ClassAds verdeutlichen (nach [Con07]).

Listing 2.1 Condor Job ClassAd

```
Executable          = foo
Requirements        = (Memory >= 32) && (OpSys == "SOLARIS28") &&
                      (Arch == "SUN4u")
Rank                = Memory >= 64
Image_Size          = 28
Error               = err.$(Process)
Input               = in.$(Process)
Output              = out.$(Process)
Log                 = foo.log
Queue 150
```

Dieser Job würde aus 150 Läufen des Programms *foo* bestehen. Das Zielsystem müsste Solaris 8 auf einer Sun-Workstation sein, mit mindestens 32 MB Hauptspeicher, bevorzugt jedoch Maschinen, die mehr als 64 MB Hauptspeicher besitzen, sofern verfügbar. Das Makro *$(Process)* bei den Error-, Input- und Output-Angaben bewirkt, dass für Job 0 die Dateien *err.0*, *in.0* und *out.0* (für *stderr*, *stdin* und *stdout*) angegeben sind, für Job 1 entsprechend *err.1*, *in.1* und *out.1* usw. Das Programm würde während der Ausführung 28 MB Hauptspeicher benutzen und Condor soll alle Informationen über die Ausführung in die Logdatei *foo.log* schreiben. Die Beschreibung einer Maschine, die diesen Job starten könnte, würde z. B. wie in Listing 2.2 aussehen:

Listing 2.2 Condor Machine ClassAd

```
MyType              = "Machine"
TargetType          = "Job"
Memory              = 1024
Arch                = "SUN4u"
OpSys               = "SOLARIS28"
State               = "Unclaimed"
HasFileTransfer     = True
Start               = (LoadAvg < 10) || (KeyBoardIdle > 15 * 60)
Requirements        = Start
Rank                = Image_Size < 500
```

Eine wichtige Rolle für die Verteilung spielen die Angaben *Start* (nur bei Rechnern), *Requirements* und *Rank*:

- *Start* beschreibt die Bedingung, die erfüllt sein muss, damit der Rechner Jobs ausführt. Im Beispiel müsste die Rechnerlast unter 10 liegen oder die Tastatur mindestens 15 Minuten nicht benutzt worden sein.
- *Requirements* wird bei der Zuordnung von Jobs zu Maschinen benutzt. Die Spezifikationen eines Jobs werden gegen die Zustandsdaten eines Rechners evaluiert. Sollte die gesamte Auswertung *true* ergeben, kann der Job auf der entsprechenden Maschine ausgeführt werden.
- *Rank* beschreibt die bevorzugte Ausführumgebung, wenn für einen Job mehrere Rechner oder für einen Rechner mehrere Jobs verfügbar sind. Der Job im obigen Beispiel würde Rechner bevorzugen, die mehr als 64 MB Hauptspeicher haben, während der Rechner im Beispiel Jobs bevorzugt, die nicht mehr als 500 MB Hauptspeicher verbrauchen.

Ein weiterer Mechanismus, der die Zuordnung von Jobs zu Rechnern beeinflusst, sind Benutzerprioritäten, die allerdings nicht vom Benutzer festgelegt, sondern vom System zur Laufzeit ermittelt werden. Ein Nutzer wird beim Verteilen von Jobs umso stärker bevorzugt, je niedriger sein Prioritätswert ist. Dieser Wert leitet sich direkt aus der Anzahl der laufenden Jobs eines Benutzers ab (je mehr Jobs, desto höher). Die Prioritätswerte und die Anzahl der Rechner, die einem Benutzer zugeteilt werden, verhalten sich invers: Ein Benutzer mit einem im Vergleich zu einem anderen Benutzer doppelt so hohen Prioritätswert erhält die Hälfte der Ressourcen als der konkurrierende Benutzer. Damit wird eine Gleichbehandlung erreicht, sofern es im Rahmen des oben beschriebenen Zuordnungsverfahrens durch ClassAds möglich ist.

Anforderungen ans Netzwerk

Wie aus den obigen Ausführungen ersichtlich wird, sind die Rechner eines Condor-Pools sehr eng gekoppelt: Ausführende Rechner stehen – vor allem bei Jobs im Standard-Universum – in permanentem Kontakt mit den Rechnern, von denen aus der Job ins System eingegeben wurde. Ein zuverlässiges, schnelles Netz ist Voraussetzung, vor allem, wenn eine Submit Machine sehr viele Jobs gleichzeitig verteilt. Die Verteilung selbst erfolgt bei Condor nach dem push-Prinzip: Die TCP-Verbindung wird vom Zugangsrechner aus zum ausführenden Rechner hin aufgebaut. Dabei werden mehrere TCP-Verbindungen benutzt, deren Portnummern teilweise erst zur Laufzeit ausgehandelt werden. Diese Tatsache erschwert den Betrieb von Condor-Rechenknoten hinter Firewalls und NAT-Routern erheblich.

2.4.2 BOINC

Bei der Berkeley Open Infrastructure for Network Computing (BOINC) [And04] handelt es sich um ein quelloffenes Framework zur Verteilung rechenintensiver Probleme auf beliebige Internet-PCs. Das Framework entstand auf Grundlage des bekannten SETI@home Projektes [ACKL+02], hat aber zum Ziel, nicht nur auf ein einzelnes Projekt festgelegt zu

sein, sondern eine Infrastruktur zu schaffen, mit dem sich beliebige Public-Ressource-Computing-Projekte aufsetzen lassen. BOINC wird inzwischen von vielen Projekten benutzt, neben SETI@home seien an dieser Stelle noch Einstein@home [Ein07] und Climaprediction.net [Cli07] genannt.

Architektur und Arbeitsweise

BOINC besteht einerseits aus einigen Komponenten, die für jedes Projekt identisch sind (der Kernclient und verschiedene Serverkomponenten) und andererseits aus Teilen, die projektspezifisch angepasst oder implementiert werden müssen (Abb. 2.6).

Der Kernclient kontaktiert den Scheduling-Server, erhält von diesem eine Arbeitspaketdefinition zugeteilt und lädt die darin angegebenen Daten vom Datenserver herunter. Im Anschluss wird ein projektspezifisches Programm gestartet, das die eigentliche Berechnung durchführt, und überwacht. Programm und Kernclient sind dabei über die BOINC-API (Application Programming Interface) gekoppelt, so dass es z. B. möglich ist, das Programm zu pausieren oder anwendungsspezifische Graphiken im Rahmen eines Bildschirmschoners darzustellen. Wurde die Berechnung durchgeführt, überträgt der BOINC-Client die Ergebnisse wieder zum Server. Der Kern-Client ist dabei nicht auf ein einzelnes Projekt beschränkt, sondern kann (auch gleichzeitig) an mehreren BOINC-Projekten teilnehmen.

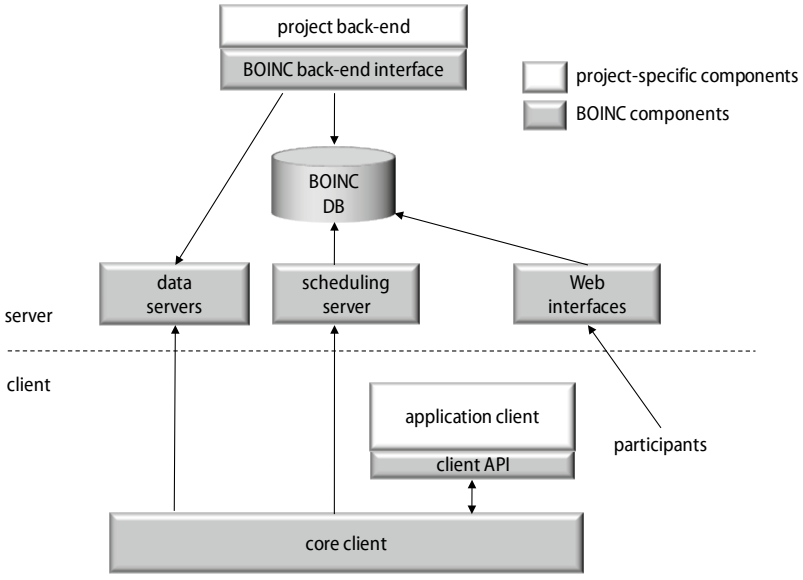


Abb. 2.6 BOINC Architektur[5]

Um unempfindlich gegenüber fehlerhaften oder gefälschten Berechnungsergebnissen zu sein, liefert BOINC jedes Arbeitspaket in mehreren Instanzen an mehrere Clients aus, von deren Ergebnissen dann dasjenige als das Korrekte gewertet wird, das von der Mehrheit der Clients erzeugt wurde.

[5] nach [Boi07]

Server

Die Serverseite eines BOINC-Projektes besteht aus mindestens einem Webserver, der den gesamten Datentransfer vom und zum Client abwickelt, und einer Datenbank, mit der die Arbeitspakete und deren Ergebnisse verwaltet werden. Auf dem zentralen Server arbeiten mehrere Dienste, die unter anderem die folgenden Aufgaben erfüllen:

- Zustandsverwaltung und Überwachung des Arbeitsfortschritts der einzelnen Arbeitspakete und darauf basierend die Entscheidung, ob ein Arbeitspaket validiert werden soll oder ob neue Instanzen des Paketes zur Berechnung erzeugt werden müssen. Die Zustandsverwaltung ist integraler Bestandteil von BOINC.
- Laden der Arbeitspaketdaten in den Hauptspeicher zur schnelleren Bearbeitung und Auslieferung an die Clients. Diese Funktion dient ausschließlich der Leistungssteigerung des Schedulers und ist ebenfalls Bestandteil des Frameworks.
- Der Scheduler entscheidet bei Anfrage eines Clients, welche Jobs von diesem berechnet werden sollen. Dabei werden Anforderungen des Clients z. B. bezüglich Speicher und Festplattenbedarf des Jobs sowie das Zielbetriebssystem berücksichtigt. Der Scheduler ist als FastCGI-Programm [Bro96] realisiert und ebenfalls Teil des BOINC-Frameworks.
- Validierung der Ergebnisse: Hier wird entschieden, ob die bearbeiteten Instanzen eines Arbeitspaketes genug gleichartige Ergebnisse erzeugt haben, um das „korrekte“ Ergebnis festzulegen. Die Validierungsfunktion ist projektspezifisch und muss daher vom BOINC-Nutzer implementiert werden.
- Nachbearbeitung und Archivierung der Ergebnisse der einzelnen Arbeitspakete. Diese Funktion muss ebenfalls vom BOINC-Nutzer implementiert werden.

Die Dienste kommunizieren ausschließlich über eine gemeinsame MySQL-Datenbank miteinander, die den gesamten Zustand eines BOINC-Projektes speichert. Die Serverkomponenten laufen unter Linux oder Solaris, aufgrund des offenen Quellcodes sind jedoch auch andere Plattformen denkbar.

Client

Clientseitig besteht BOINC im Wesentlichen aus zwei Komponenten: Der Kernclient, der für alle Projekte identisch ist, und einem projektspezifischen Teil, der vom Nutzer selbst implementiert werden muss. Dem Kernclient teilt man mit, für welches Projekt man seinen PC mitrechnen lassen will, woraufhin dieser den entsprechenden Schedulingserver kontaktiert, die projektspezifische Anwendung und die zugehörigen Daten herunterlädt und die Anwendung startet. Als Teilnehmer eines BOINC-Rechnerverbundes kann man Richtlinien bezüglich der vom Client durchgeführten Jobs vergeben: Beispielsweise ist es möglich, für mehrere Projekte gleichzeitig zu arbeiten und die verwendete CPU-Zeit individuell zwischen den einzelnen Projekten aufzuteilen. Weiterhin kann man die maximale lokale Ressourcennutzung (z. B. verwendeter Festplattenplatz) durch die Jobs festlegen. Der Kernclient ist für viele verschiedene Betriebssysteme wie Windows, Linux oder Mac OS verfügbar.

Die eigentliche Rechenanwendung, die der Nutzer implementiert, muss als ausführbare Datei vorliegen und am BOINC-Server registriert sein. Anschließend kann der Nutzer Arbeitspakete für sein Projekt definieren: Ein Arbeitspaket besteht aus einer Menge von Dateien und Kommandozeilenargumenten für die registrierte Anwendung. Sie muss mit der BOINC-C-API verlinkt werden und von dieser einige Funktionen aufrufen. Weiterhin kann die Anwendung dem BOINC-Kernclient den Berechnungszustand mitteilen, ebenso ist das Übermitteln von Bildschirmschonergraphiken (zur Unterhaltung des Clientbetreibers) möglich. Ist die Applikation so implementiert, dass sie in regelmäßigen Abständen ihren eigenen Zustand persistent speichern und auf diesem Zustand wieder aufsetzen kann, kann sie sich durch einen API-Aufruf vom Client beenden und wieder neu starten lassen. Dieses „Checkpointing“ muss allerdings vom Nutzer selbst implementiert werden. BOINC-Anwendungen werden normalerweise in C/C++ oder Fortran geschrieben, für Java und Python existieren jedoch Wrapper.

Verteilverfahren

Das Verfahren, nachdem BOINC Jobs an die einzelnen Rechenknoten ausliefert, ist im Wesentlichen clientseitig realisiert: Der Kernclient entscheidet, wann er arbeiten möchte, für welches Projekt (also für welchen Nutzer) er arbeiten möchte und wie viele der lokalen Ressourcen welchem Projekt zugeteilt werden. Dies alles geschieht nicht automatisch, sondern muss vom Betreiber eines Clients festgelegt werden. Serverseitig findet ein Scheduling praktisch nur auf der Ebene statt, dass einem Nutzer nicht mehrfach eine Instanz des gleichen Arbeitspakets zugeteilt wird und dass die Anforderungen des Clients bezüglich der Ressourcennutzung und natürlich des Ziel-Betriebssystems berücksichtigt werden.

Anforderungen ans Netzwerk

BOINC als klassisches Client/Server-System (vgl. Abschnitt 2.1.1) stellt nur wenige Anforderungen an die Netzwerktopologie: Die Clients kontaktieren den Server aus eigenem Antrieb, die TCP-Verbindung wird ausschließlich vom Client initiiert. Deswegen funktioniert BOINC im Gegensatz zu Condor problemlos hinter NAT-Routern im Internet. Auch sonstige Firewalls verursachen keinerlei Probleme, sofern das (bei BOINC ausschließlich verwendete) HTTP-Protokoll nicht blockiert wird.

2.4.3 Alchemi

Alchemi [LBRV05, Alc07] wird seit 2003 im Rahmen des Gridbus Projektes der Universität von Melbourne [GBus07] entwickelt und von vielen Institutionen eingesetzt. Es handelt sich um ein recht neues Verteilsystem, das speziell zum Einsatz auf Desktop-PCs entwickelt wurde und sich in zwei wesentlichen Punkten von anderen Clustersystemen unterscheidet: Es basiert in allen Komponenten auf Windows und die Benutzung folgt einem objektorientierten Ansatz, der sich an die Thread Programmierung nicht-verteilter Anwendungen anlehnt. Alchemi wurde auf der Microsoft .NET Plattform in C# implementiert, was es ebenfalls von den meist in C/C++ geschriebenen üblichen Verteil- und Clustersystemen unterscheidet.

Architektur und Arbeitsweise

In einer Alchemi-Installation existieren drei verschiedene Rollen, die die beteiligten Rechner einnehmen müssen: Der Rechner des Entwicklers, auf dem der sequentielle Teil der parallelen Anwendung läuft, ein Managerknoten, der die parallelen Teile der Anwendung verwaltet, und mehrere Ausführungseinheiten, auf denen der parallele Code letztlich ausgeführt wird (siehe Abb. 2.7). Die Kommunikation basiert auf dem .NET Remoting-Mechanismus, der, vergleichbar mit Java RMI, Methodenaufrufe entfernt ausführt und dabei komplexe Objekte über das Netzwerk transportieren kann (vgl. 2.2.2).

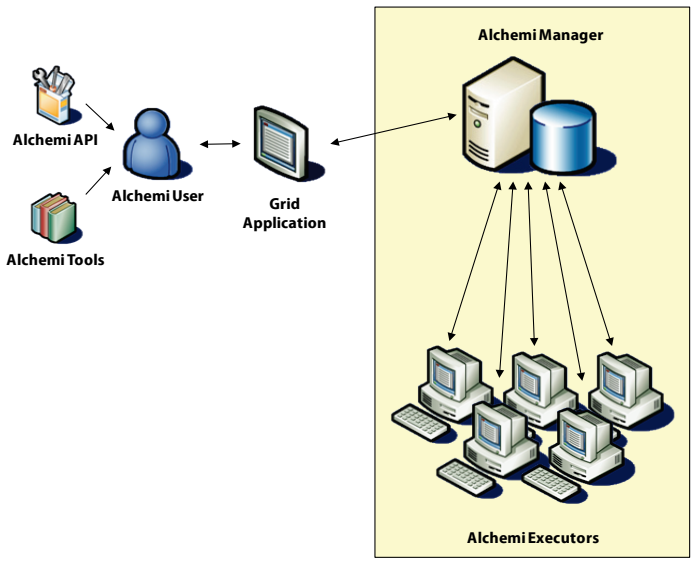


Abb. 2.7 Rollen in Alchemi[6]

Komponenten

Wie oben bereits erläutert, existieren in einem Alchemi-Cluster verschiedene Rollen, die im Folgenden genauer erläutert werden.

Benutzerrechner: Hierbei handelt es sich um die Rechner der Nutzer/Entwickler paralleler Anwendungen. Von hier aus wird der parallele Teil einer Anwendung an den Manager übertragen, der sequentielle Teil wird auf dem Benutzerrechner ausgeführt: Der Nutzer übergibt hier die Jobs per Kommandozeilentools oder durch Nutzung der GridThread-API an das Alchemi-System.

Manager: Dabei handelt es sich um den zentralen Server des Systems, der über eine Remoteschnittstelle die parallelen Objekte entgegennimmt, in eine Warteschlange einreiht und an die verfügbaren Ausführungsrechner zur Berechnung verteilt. Die einzelnen Tasks können prioritätsbasiert oder auf „First Come First Serve"-Basis verteilt werden, wobei die Priorität vom Nutzer angegeben werden muss. Weiterhin existiert eine SOAP-Webservice-Schnittstelle, mit der es möglich ist, von beliebigen SOAP-fähigen Anwendungen aus Jobs an das System zu übermitteln. Der Betrieb des Managers setzt eine laufende MS-SQL Server Datenbank voraus.

[6] Nach [Alc07]

Ausführungsknoten: Ein Ausführungsrechner kann in zwei verschiedenen Modi betrieben werden: dediziert und auf freiwilliger Basis. Als dedizierter Rechenknoten bietet der Rechner eine Remoteschnittstelle an, die bei Bedarf vom Manager aus direkt angesprochen wird. In diesem Fall entscheidet also der Manager, was der Knoten tut und wann er es tut. Somit ist eine flexible Verwaltung der Rechenressourcen möglich. Betreibt man einen Rechenknoten auf freiwilliger Basis, verhält er sich vergleichbar mit einem BOINC-Client: Der Betreiber des Rechenknotens entscheidet, wann der Knoten beim Manager nach Jobs anfragt. Derart betriebene Rechner können leicht hinter NAT-Routern oder sonstigen Firewalls betrieben werden, da der TCP-Verbindungsaufbau immer zum Manager hin initiiert wird.

Anwendungsmodelle

Alchemi-Anwendungen folgen dem Master-Worker-Anwendungsmodell, bei dem ein sequentieller Programmteil die parallel auszuführenden Teile startet, auf deren Ende wartet und dann die Einzelergebnisse zu einem Gesamtergebnis kombiniert. Als Nutzer von Alchemi kann man dabei auf zwei Möglichkeiten zurückgreifen: über die .NET-basierte GridThread-API oder mit klassischen dateibasierten Jobs. Die beiden Varianten unterscheiden sich nur in der Art und Weise, wie man das System anspricht und die parallelen Tasks an den Manager übermittelt, bei der eigentlichen Verteilung auf die Rechenknoten passiert in beiden Fällen das gleiche: Per .NET Remoting werden Objekte serialisiert und zu den Rechenknoten übertragen, der Code in den Objekten ausgeführt und anschließend wird das Objekt, aus dem jetzt das Ergebnis ausgelesen werden kann, ebenfalls per Remoting zurückübertragen (Abb. 2.8).

Grid Thread Model: Hier verteilt man die Anwendung, indem man den parallel auszuführenden Code in frei parametrisierbare Klassen auslagert, die man dann instanziiert, startet und sich über deren Ausführungsende informieren lässt. Die Alchemi .NET-API versteckt die Verteilung und Remoteausführung der Objekte und dem dort spezifizierten Code vollständig vor dem Entwickler. Die Parallelisierung einer Anwendung entspricht damit im Wesentlichen der üblichen lokalen Threadprogrammierung. Den Nutzern stehen dabei alle objektorientierten Konzepte wie Kapselung, Vererbung, Ereignisse etc. sowie alle Programmiersprachen, für die ein .NET-Compiler existiert, zur Verfügung.

Grid Job Model: Hat man eine Anwendung zu verteilen, die nur in einer fertig kompilierten ausführbaren Datei zur Verfügung steht, nutzt man das Grid Job Modell. Der Nutzer definiert die einzelnen Jobs in einem XML-Dokument, indem Befehlszeile, Quell- und Ergebnisdateien angegeben und diese Jobbeschreibungen anschließend mit einem Kommandozeilentool an den Alchemi-Manager übergeben werden.

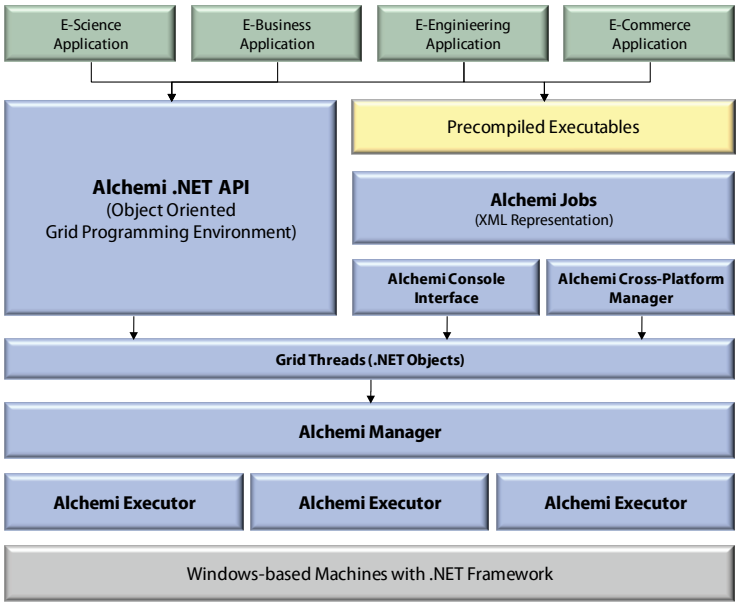


Abb. 2.8 Alchemi Architektur und Anwendungsmodell[7]

Verteilverfahren

Die Verteilung erfolgt bei Alchemi zentral vom Manager aus gesteuert: Der Server ist die zentrale Komponente, die entscheidet, wann ein Job auf welchem Knoten ausgeführt wird. Dabei werden lediglich die von jedem Benutzer einstellbaren Prioritäten beachtet, ansonsten wird ein einfaches First Come First Serve verwendet, das die Leistungsfähigkeit der einzelnen Rechner natürlich nicht berücksichtigen kann. Wie Condor geht man nämlich auch bei Alchemi davon aus, dass ein Rechner einen an ihn zugewiesenen Job korrekt und zuverlässig ausführt.

Anforderungen ans Netzwerk

Die Kommunikation zwischen Entwicklerrechner und Manager wird bei Alchemi immer vom Entwicklerprogramm aus angestoßen, die Ausführungseinheiten können jedoch auch explizit vom Manager aus angesprochen werden. Alternativ initiieren sie die Kommunikation zum Manager selbst, indem der TCP-Verbindungsaufbau vom Rechenknoten aus erfolgt. Damit kann das System problemlos hinter NAT-Routern und Firewalls betrieben werden, lediglich der Manager muss auf den benutzten TCP-Ports (Remoting und HTTP für die SOAP-Schnittstelle) erreichbar sein.

2.4.4 Zusammenfassung und Vergleich

Mit Condor, BOINC und Alchemi wurden drei Systeme zur Verteilung rechenintensiver Tasks auf (Standard-)PCs vorgestellt. Condor und Alchemi eignen sich zum Einsatz in Rechnerpools besser als BOINC, da das Aufsetzen einer Anwendung bzw. eines Projektes in BOINC einen gewissen Aufwand erfordert: Die Serverinfrastruktur mit den verschiedenen Webservern und der Datenbank muss für jedes Projekt extra eingerichtet werden.

[7] Nach [LBRV05]

Nicht zu vergessen ist, dass die Anwendung noch an BOINC angepasst und mit der API verlinkt werden muss, was für nicht in C/C++ geschriebene Programme teilweise schwierig oder gar nicht realisierbar ist. Dieser hohe Aufwand lohnt sich also nur, wenn man ein derartig lang laufendes Projekt mit mehreren tausend Jobs hat, so dass die Verteilung an Freiwillige im Internet sinnvoll ist. Sollen aber viele Nutzer mit möglicherweise sehr unterschiedlichen Anforderungen ad hoc ihre Jobs an ein Verteilsystem übergeben können, sind Systeme wie Condor und Alchemi besser geeignet.

Condor ist sehr flexibel beim Zuteilen von Jobs an Rechenknoten was die unterschiedliche Ausstattung der Rechner mit Hardware und Betriebssystem angeht, ist jedoch bei der eigentlichen Jobfunktionalität sehr Unix- und C-lastig: Benutzt man Java oder kommt auf den Rechenclients Windows zum Einsatz, besteht z. B. keine Möglichkeit mehr, transparentes Checkpointing zu benutzen. Ein wesentlicher Nachteil von Condor ist die enge Kopplung der einzelnen Rechner und die Tatsache, dass die Rechner nicht hinter NAT-Routern betrieben werden können. Die Benutzung des Systems mit Kommandozeilenwerkzeugen und die Erstellung einer komplexen ClassAd-Datei pro Job sind nicht sehr benutzerfreundlich.

Die Stärke von Alchemi liegt eindeutig in der objektorientierten API, man ist jedoch bei deren Benutzung – wie bei Programmierschnittstellen allgemein typisch – auf die Verwendung einer bestimmten Sprache angewiesen: Im Falle von Alchemi sind das .NET-Sprachen. Benutzt man Alchemi, um (programmiersprachenunabhängige) Nicht-API-Jobs zu verteilen, ist man auf die Erstellung von XML-Dokumenten angewiesen, die ebenfalls mit Kommandozeilentools an das System übergeben werden.

Bei den Verteilverfahren und der Toleranz gegenüber fehlerhaft oder unzuverlässig arbeitenden Clients existieren ebenfalls Unterschiede: BOINC überlässt es dem Client, zu entscheiden, wann für welches Projekt gearbeitet wird und wie die lokale Rechenzeit auf einzelne Projekte verteilt wird. Dabei verhält sich BOINC sehr fehlertolerant: Hat der Nutzer Checkpointing in seiner Anwendung implementiert, kann der Client die Berechnung zu praktisch beliebigen Zeitpunkten oder nach einem Systemabsturz am letzten Checkpoint fortsetzen. Condor und Alchemi besitzen ebenfalls Mechanismen, die einen Ausfall eines Clients erkennen und den dort laufenden Job auf einem anderen Rechner neu starten. Bei Condors Standardjobs greift ebenfalls ein Checkpoint-Mechanismus, den der Nutzer hier im Gegensatz zu BOINC nicht selbst implementieren muss. Weitergehende Entscheidungen bezüglich des Schedulings treffen die Systeme allerdings nicht: Condor verteilt die Jobs aufgrund der ClassAD-Beschreibung von Job und Rechner, verlässt sich allerdings darauf, dass die Rechner sich entsprechend der Beschreibung verhalten und zuverlässig arbeiten, BOINC-Scheduling findet ausschließlich beim Client statt und Alchemi geht wie Condor zunächst davon aus, dass ein Rechner einen Job auch zu Ende ausführt. Informationen über die reale Leistungsfähigkeit der Clients, deren Zuverlässigkeit und die tatsächliche Laufzeit der Jobs werden jedoch bei keinem der drei Verteilverfahren berücksichtigt. Ebenso wenig finden sich Ansätze oder Versuche, das Laufzeitverhalten, wie z. B. die Zuverlässigkeit, der Clients zu analysieren [BSV03, KCBW02], zu quantifizieren [BNW03] oder gar vorherzusagen und – wie etwa in [BKCB+06, MiNo06] vorgeschlagen – dieses Wissen in die Verteilung der Jobs einfließen zu lassen.

2.5 Gridcomputing

Die in den vorangegangenen Abschnitten beschriebenen Verteilsysteme haben – so unterschiedlich sie auch sind – eines gemeinsam: Ein Berechnungsauftrag, der an eines dieser Systeme übergeben wurde, wird von genau diesem System bzw. den daran angeschlossenen Rechenknoten ausgeführt. Ein Job, den man beispielsweise an den HP XC4000 Höchstleistungsrechner des Karlsruher Rechenzentrums[8] zur Berechnung übergibt, wird niemals von einem BOINC-Client ausgeführt, auch wenn dieser vielleicht technisch dazu in der Lage wäre. Umgekehrt würde ein SETI@home Arbeitspaket nicht auf einem dieser Großrechner ausgeführt werden können. Die formale Beschreibung der Jobs, die Zugangsschnittstellen zu den Systemen, die technischen Gegebenheiten auf den Rechenknoten, die administrativen und netztechnischen Randbedingungen, sowie die Abrechnungsmodelle (sofern vorhanden) sind zu unterschiedlich. Diese Schnittstellen so zu vereinheitlichen, dass Rechnerressourcen universell verfügbar und nutzbar sind, ist das Ziel des Gridcomputing, wie es unter anderem in [FoKe99] beschrieben wird:

> *"The grid will connect multiple regional and national computational grids to create a universal source of computing power."*

Der Begriff „Grid" wird dabei analog zum Stromnetz gewählt (engl. *electric power grid* 'Stromnetz'): Man verfügt über einen einfachen, universell zu nutzenden Zugriff auf Rechnerressourcen, welche nicht zwingend reine Rechenleistung sein müssen, auch Daten- oder Wirtschaftsdienste sind denkbar. Die Definition hat man später noch verfeinert, indem drei zentrale Eigenschaften, die ein Gridsystem aufweisen muss, festgelegt wurden [Fos02]:

- Die von einem Grid koordinierten Ressourcen unterliegen keiner zentralen Kontrolle, sondern können sowohl geographisch, funktional, als auch administrativ verteilt sein.
- Ein Grid verwendet ausschließlich offene, standardisierte Protokolle und Schnittstellen, um grundlegende Angelegenheiten, wie z. B. Authentifizierung oder Ressourcenzugriff, zu regeln.
- Ein Grid hat die Fähigkeit, die von ihm angebotenen Dienste so zu koordinieren, dass deren Nutzung unter Einhaltung verschiedenster Dienstgütekriterien ermöglicht wird.

Cluster- oder Supercomputermanagementsysteme wie die Sun Grid Engine oder das Portable Batch System (vgl. Abschnitt 2.3) sind demnach keine Grids. Genauso wenig handelt es sich bei BOINC-Projekten um Gridcomputing. Für Condor existiert allerdings eine Griderweiterung [FTFL+02], mit der Condorjobs an Rechensysteme weitergeleitet werden können, die eine Globus-kompatible Schnittstelle besitzen. Um Griddienste zu implementieren bzw. vorhandene Systeme um Gridfähigkeiten zu erweitern, wird meist UNICORE [ErSn01] oder das bereits erwähnte Globus Toolkit [Fos06] verwendet, eine komplexe Sammlung von Protokollen und Frameworks, die unter dem Global Grid Forum als Open Grid Services Infrastructure/Architecture (OGSI/OGSA) weiterentwickelt

[8] http://www.rz.uni-karlsruhe.de/ssck/hpxc4000.php

werden. Ein Ziel dabei besteht darin, die Technik der Webservices (vgl. Abschnitt 2.2.2) zu integrieren, wobei das verwendete Web Service Ressource Framework (WSRF) die für ein Grid System wichtige Zustandsverwaltung für Webservices definiert [FCFF+05].

Verschiedenste Projekte widmen sich dem Aufbau von Gridinfrastrukturen, als Beispiel seien hier die D-Grid Initiative [NeKe07], die verschiedenste Projekte im Bereich der Forschung, Industrie und Wirtschaft vereint, und das LHC-Grid [LHC08], das die Daten von Teilchenbeschleuniger-Experimenten auswertet, genannt.

2.6 Zusammenfassung

Ausgehend von der Beschreibung grundlegender Architekturen verteilter Systeme wurden zunächst wichtige Basistechnologien des Internets vorgestellt, soweit sie für das technische Verständnis der weiteren Arbeit notwendig erscheinen. Neben verbreiteten Basisprotokollen des Internets wurden Konzepte zur Realisierung verteilter Anwendungen erläutert, wie sie in einem System zur Rechenjobverteilung zum Einsatz kommen können. Im weiteren Verlauf wurden Techniken zur Realisierung von parallelen nachrichtengekoppelten Programmen für Höchstleistungsrechner und spezialisierten Rechenclustern vorgestellt, sowie ein kurzer Überblick in den Aufbau und die Nutzung von Supercomputern gegeben.

Ein für diese Arbeit jedoch weit wichtigerer Aspekt als Höchstleistungsrechner sind Systeme, die rechenintensive Aufgaben Internet-basiert auf normale Arbeitsplatzrechner verteilen, dort ausführen können, und auch üblicherweise in genau diesem Szenario genutzt werden. Das Arbeitsplatz- und Pool-Einsatzgebiet stellt auch das des im weiteren Verlauf der Arbeit vorgestellten JoSchKa-Systems dar: Es ist mit Condor, BOINC und Alchemi vergleichbar und damit klar gegen dedizierte Rechencluster und Höchstleistungsrechner abgegrenzt. Der Abschnitt über Gridcomputing soll vor allem klären, was Gridcomputing darstellt, und was man darunter nicht (oder nicht mehr) versteht.

Das im Rahmen dieser Arbeit ab Kapitel 3 beschriebene und realisierte System verfolgt einen ähnlichen Architekturansatz wie BOINC und Alchemi (eine Client/Server Architektur, bei der Standardinternetprotokolle und Datenformate eingesetzt werden), verhält sich aber generischer und ist für Entwickler paralleler Anwendungen ähnlich schnell ad hoc nutzbar wie ein Condor-Cluster. Anwenderfreundliche graphische Tools zum Erstellen und Verwalten der Jobs sowie eine API zur programmgesteuerten Nutzung und Verteilung parallel ausführbarer Codeblöcke wurden ebenfalls realisiert. Bei den eingesetzten Verfahren zur Entscheidung, welcher Job auf welchem Rechner ausgeführt wird, verfolgt JoSchKa jedoch einen völlig anderen Ansatz, der die speziellen Gegebenheiten beim Verteilen von Jobs auf Rechnerpools und Arbeitsplatzrechnern berücksichtigt.

Kapitel 3

Entwurf von JoSchKa

Dieses Kapitel beschreibt den Entwurf und die Architektur eines Systems zum Verteilen von rechenintensiven Jobs auf beliebige Rechenknoten: JoSchKa [BTS05]. Der Fokus liegt im Besonderen darauf, dass kein spezialisierter Rechencluster aufgebaut werden soll, sondern ein System entsteht, durch das beliebige Arbeitsplatzrechner zu einem losen Rechenverbund zusammengeschlossen werden können. Die Ziele wurden wie folgt definiert:

- autonome Rechenknoten, Unterstützung heterogener Systeme und beliebiger Programmiersprachen, sofern diese auf dem Zielsystem lauffähig sind,
- Datenaustausch nicht über ein gemeinsames Dateisystem,
- vielfältige variable Einsetzbarkeit, sowie Nutzung bestehender Eigenentwicklungen bei minimalem Einarbeitungsaufwand für Nutzer,
- minimaler Installations- und Wartungsaufwand bei den Rechenknoten und
- faire adaptive Verteilung der Jobs aller beteiligten Nutzer unter Berücksichtigung eventueller Heterogenität und Unzuverlässigkeit der einzelnen Rechenknoten.

Der hier beschriebene Ansatz folgt der Idee, dass ein zentraler Server die vom Nutzer dort hinterlegten Jobs verwaltet und ein auf jedem Rechenknoten laufendes Programm (im Folgenden auch *Agent* genannt) selbsttätig beim Server anfragt, ob es etwas zu bearbeiten gibt. Der zugeteilte Rechenjob wird daraufhin vom Server heruntergeladen, ausgeführt und danach das Ergebnis wieder zum Server übertragen. Der Nutzer holt anschließend beim Server die Ergebnisse ab (siehe Abb. 3.1).

3.1 Rechenjobverteilung mit Webservices

Das Grundkonzept des hier beschriebenen Systems liegt darin, dass ein Server existiert, der ein Portfolio an Rechenjobs besitzt, aus denen er einen geeigneten, zur jeweiligen Anfrage des Agenten passenden, auswählt (vergleichbar mit den Tuplespaces aus z. B. [Gel85]). Deshalb müssen sich die einzelnen Jobs ausreichend gut selbst beschreiben. Der Agent spezifiziert beim Anfragen genau, welche Art von Jobs er ausführen will (z. B. „nur Jobs von Benutzer A.“) oder gibt technische Randbedingungen (z. B. „keine Jobs die Java benötigen“) an. Weiterhin sollte ein Agent in der Lage sein, auf seiner Plattform beliebige Kommandos auszuführen (also Programme oder Skripte starten), beliebige Dateien von und zu einem Server zu übertragen, und dabei völlig autonom agieren können. Die zwi-

schen Agent und Server ausgetauschten Status- und Steuerinformationen werden mit Webservicetechnologien (siehe Kapitel 2.2.2) übertragen. Der Server besitzt eine SOAP-Schnittstelle, Agent und Server kommunizieren nur für kurze Zeit miteinander und der Datenaustausch wird dabei grundsätzlich vom HTTP-Client des Agenten initiiert. Es sind also keine Netzwerkdienste auf dem Rechenknoten notwendig.

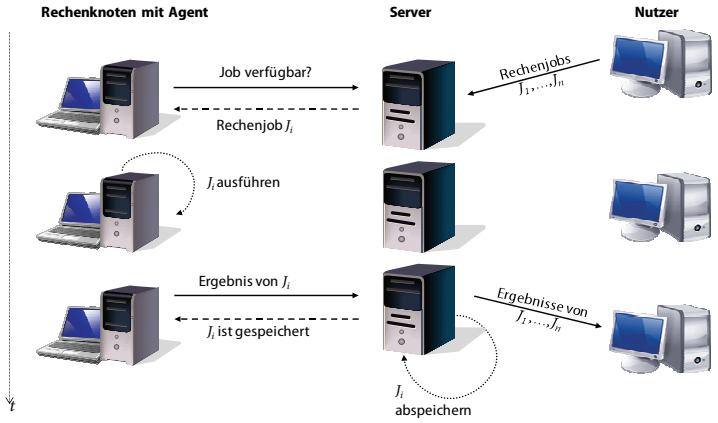


Abb. 3.1 JoSchKa Arbeitsprinzip

Im Folgenden wird auf der grundsätzlichen Systemarchitektur basierend beschrieben, welche Anforderungen an die einzelnen Komponenten gestellt werden, wie sie aufgebaut sein müssen, welche Datenmodelle zum Einsatz kommen und wie die einzelnen Schnittstellen des Servers aufgebaut sind. Zusätzlich wird die Arbeitsweise der einzelnen Komponenten detailliert dargestellt. Ein Abschnitt zu Sicherheitsfragen schließt das Kapitel ab.

3.2 Architektur

Wie im vorangegangenen Abschnitt erläutert, muss der Server einzelne Rechenjobs verwalten. Ein solcher Job wird durch die folgenden wesentlichen Eigenschaften charakterisiert:

- eine Klassifizierung, die den Job einem Benutzer und innerhalb der Menger aller Jobs eines Benutzers einem Projekt oder Typ zuordnet,
- eine innerhalb der Jobmenge eindeutige Identifikationsnummer,
- eine Statusbeschreibung (frei, laufend, fertig, …),
- das vom Agenten auszuführende Kommando,
- Listen von Quell- und Ergebnisdateien und
- statistische Daten über verbrauchte Rechenzeit sowie den ausführenden Rechenknoten.

Tabelle 3.1 zeigt exemplarisch und vereinfacht, wie eine solche Jobbeschreibung aufgebaut ist. Diese Daten können z. B. in einem relationalen Datenbanksystem oder als XML-Dokument gespeichert werden.

Tabelle 3.1 Vereinfachte Jobbeschreibung

jobType	jobID	status	command	files	resultFiles	node	time
csc_select	001	DONE	cmd.exe /c select.exe -v	Select.exe	out.log	172.22.131.42	1:23:45.6
mbo_bench	002	FREE	bench.exe a 1 1	Bench.exe	result.log		
alba_diplom	003	WORKING	java –jar tsp.jar –p=1	tsp.jar	r1.zip;r2.zip	172.22.126.33	runs 20 min

Der Server besitzt mehrere, meist HTTP-basierte Schnittstellen zur Kommunikation mit den Agenten auf den Rechenknoten. Hinzu kommen ein Datenhaltungsteil, der die einzelnen Jobs verwaltet, und verschiedene Managementfunktionen, die der Benutzer- und Rechenknotenverwaltung dienen.

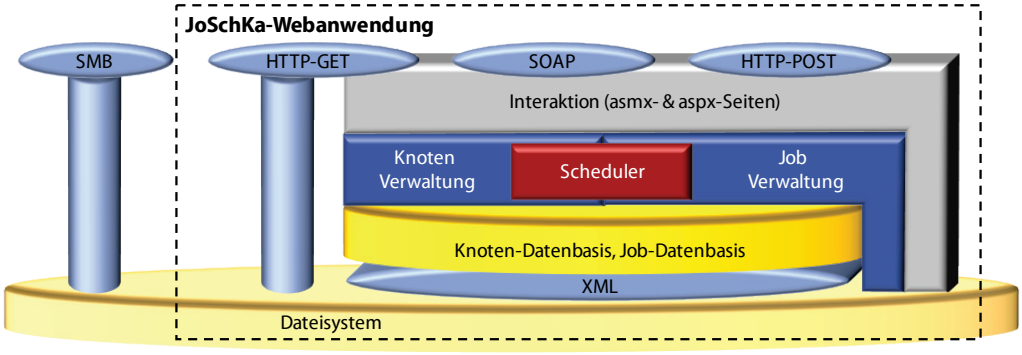


Abb. 3.2 Server Architektur

Die einzelnen Serverkomponenten sowie ihre primären Funktionen können Abb. 3.2 entnommen werden. Die wichtigste Aufgabe übernimmt dabei die Jobverwaltung. Diese Schicht realisiert die Abbildung der HTTP-POST und SOAP-Anfragen auf die interne Jobrepräsentation und die persistente Speicherung dieser Daten (sofern kein Datenbanksystem zum Einsatz kommt). Eine Vielzahl unterschiedlicher Verwaltungsfunktionen sowie diverse Fehlererkennungsmechanismen sind hier implementiert. Ein konfliktfreier Parallelzugriff auf den Datenbestand und eine konsistente Speicherung der Jobdaten werden gewährleistet. Für weitere Details zum Server sei an dieser Stelle auf die Abschnitte 3.3 und 4.1 verwiesen.

Die primären Aufgaben des JoSchKa-Agenten lassen sich wie folgt charakterisieren:

(1) Feststellen der Systemumgebung,

(2) Stellen einer Anfrage an den Server,

(3) falls ein Job verfügbar ist, dessen Daten herunterladen und den Job starten,

(4) während der Job läuft, periodisch im Hintergrund Statusmeldungen an den Server schicken, dabei auf das Ende des Jobs warten oder den Job gegebenenfalls abbrechen, sofern vom Server gewünscht und

(5) nach erfolgreicher Beendigung des Jobs Ergebnisdatei(en) zum Server hochladen und von vorne (2) beginnen. Falls Job abgebrochen wurde, pausieren und bei (2) beginnen.

Dieser Ablauf ist in Abb. 3.3 dargestellt. Man erkennt, dass beim Client mehrere parallele Threads zum Einsatz kommen und dass auf verschiedene Betriebssystemfunktionen zurückgegriffen wird, um die Jobs zu starten und auf deren Ende zu warten (vgl. 3.4ff).

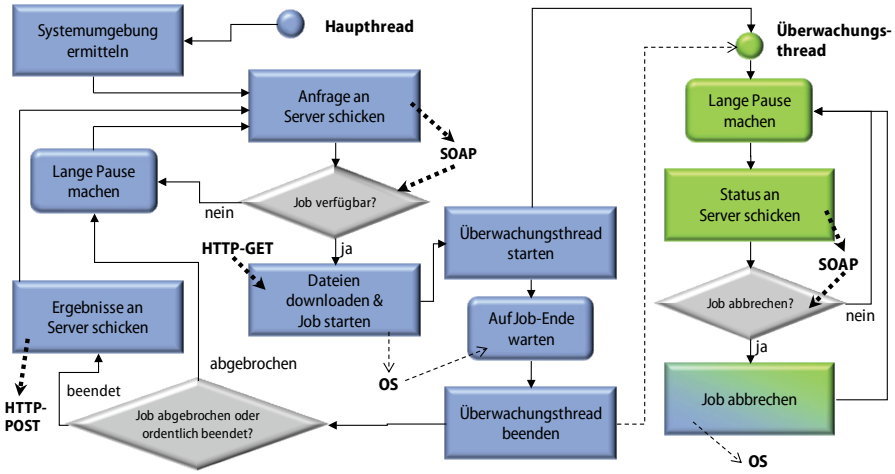


Abb. 3.3 Entwurf Agent

Ein wichtiger Aspekt in diesem Zusammenhang ist die Prozesspriorität: Ein laufender Rechenjob darf sich nur als Hintergrundlast bemerkbar machen und die Arbeit des am Rechner arbeitenden Benutzers nicht stören oder beeinträchtigen (vgl. 4.6). Aus diesem Grund startet der Agent sämtliche Jobs immer mit der niedrigsten verfügbaren Prozess-Priorität. Für weitere Details zum Agenten sei an dieser Stelle auf die Abschnitte 3.4 und 4.2 verwiesen.

3.3 Server

Beim Server handelt es sich um die zentrale JoSchKa-Komponente. Ihm obliegt die Aufgabe, die einzelnen Jobs zu speichern und den anfragenden Agenten zuzuteilen. Neben der Speicherung der Jobdaten kommen als weitere Aufgabe die Verwaltung der Quelldateien und der Berechnungsergebnisse hinzu. Der wichtigste Teil des Servers ist ein Scheduler, der entscheidet, welcher Job an einen anfragenden Rechenknoten ausgeliefert wird. Zunächst sollen jedoch die Schnittstellen für Nutzer und Agenten sowie die zentralen Datenflüsse und das Datenmodell eines Jobs beschrieben werden. Anschließend werden Fehlertoleranzmechanismen und die Verteilstrategien beschrieben. Auf die Darstellung von Implementierungsdetails wird dabei verzichtet, soweit nötig werden jedoch Datentypen, -formate und Protokolle erwähnt bzw. erläutert.

3.3.1 Schnittstellen und Datenfluss

Der Server besitzt vier Schnittstellen (von denen drei mit dem Webprotokoll HTTP angesprochen werden, siehe. Abb. 3.2), die unterschiedlichen Zwecken dienen und serverseitig unterschiedlich realisiert sind.

SOAP-Webservice: Er dient dem Austausch von Status- und Steuerinformationen zwischen Agent und Server bzw. zwischen Managementwerkzeugen und Server und bietet

die folgenden Kernfunktionen: Verwaltung der einzelnen Jobdaten, Hinzufügen und Löschen von Rechenjobs, Verwaltung und Fernsteuerung der Agenten, sowie die Abfrage von Zustandsinformationen der einzelnen Datenbestände. Über diese Schnittstelle versorgen sich die Agenten mit Rechenjobs und bestätigen deren Ausführung, sowie deren Abschluss. Angesprochen wird die Schnittstelle mit dem Webservice-Protokoll SOAP, als Transportprotokoll kommt HTTP zum Einsatz.

Quelldaten-Download: Sie dient dem effizienten Download der auszuführenden Dateien und sonstigen benötigten Dateien durch den Agenten. Verwendet wird ein normaler Webdownload mit HTTP-GET.

Ergebnisdaten-Upload: Da eine Übertragung von Binärdateien mittels SOAP sehr ineffizient ist und sowohl beim Client als auch beim Server zu beträchtlichem Umkodierungsaufwand führt (z. B. Base64-Kodierung zur ASCII-Übertragung von Binärdaten), wurde eine zusätzliche Schnittstelle realisiert, die das Übertragen von Dateien in Form eines HTTP-Uploads ermöglicht. Die Daten werden binär im POST-Teil einer Anfrage übertragen. Diese Schnittstelle existiert in zwei Varianten: Eine für die Übertragung von Berechnungsergebnissen durch den Agenten und eine für den Upload von Thread-Code-DLLs (siehe dazu Abschnitt 3.7, Thread-API).

Quell- und Ergebnisdaten für Nutzer: Da der Nutzer einer parallelen Anwendung Quelldateien am Server hinterlegen muss und seine Ergebnisdateien letztlich auch erhalten soll, wurde eine SMB-Schnittstelle (vgl. 2.2.1, SMB/CIFS) eingerichtet, durch die es für einen Anwender wesentlich bequemer möglich ist, mit einem einzelnen Arbeitsschritt viele Dateien zu übertragen, als das im Vergleich mit HTTP möglich wäre. Es ist ebenso denkbar, diese Schnittstelle z. B. mit FTP, SSH oder dem HTTP-basierten WebDAV [RFC4918] zu realisieren. Diese Schnittstelle wird nur vom Nutzer verwendet und kommt ausschließlich vor und nach der Ausführung der parallelen Anwendung zum Einsatz. Während der Jobverteilung und der Ausführung wird sie nicht benötigt.

Die Agenten auf den Rechenknoten benutzen ausschließlich die HTTP-basierten Schnittstellen und funktionieren daher in der Regel auch hinter NAT-Routern oder Firewalls. So lässt sich jeder beliebige Rechner ohne großen Aufwand an das System anbinden. Das Zusammenspiel von Agenten und Server läuft wie folgt ab:

(1) Ein Agent fragt beim Server über die SOAP-Schnittstelle nach einem Job.

(2) Der Server durchsucht seine Datenbasis nach einem, entsprechend den Anforderungen des Agenten, geeigneten Job und schickt als Antwort dessen Beschreibung.

(3) Der Agent lädt über die Download-Schnittstelle alle in der Jobbeschreibung spezifizierten Quelldateien herunter.

(4) Der Agent startet mit der in der Jobbeschreibung angegebenen Befehlszeile den Job und zusätzliche Überwachungs-Threads, die regelmäßig mit dem Server per SOAP-Schnittstelle Zustandsdaten austauschen.

(5) Nach Job-Ende überträgt der Agent alle in der Jobbeschreibung spezifizierten Ergebnisdateien sowie die Konsolenausgaben *stdout*, *stderror* und den Exit-Code der gestarteten Anwendung per Upload-Schnittstelle zum Server.

(6) Falls alle Ergebnisdateien erfolgreich beim Server gespeichert wurden, schickt der Agent eine letzte Bestätigung. Der Server markiert daraufhin den Job als endgültig abgeschlossen. Er steht erst dann für andere Agenten nicht mehr zur Verfügung.

Durch während der Ausführung regelmäßig ausgetauschte Zustandsdaten wird sichergestellt, dass der Server immer darüber informiert ist, welcher Agent gerade wie lange an welchem Job arbeitet. Sollten diese Daten einmal für längere Zeit nicht zum Server geschickt werden, wird angenommen, dass der Rechenknoten ausgefallen ist. Der entsprechende Job wird dann wieder für andere Agenten zur Ausführung freigegeben. Dieses Verfahren stellt einen zuverlässigen Betrieb auch mit stark schwankender und dynamischer Anzahl von unzuverlässigen Rechenknoten sicher (vgl. Abschnitt 3.3.3ff).

3.3.2 Jobdatenmodell

Für jeden Rechenjob werden serverseitig 22 Attribute verwaltet, die intern in einer Tabellenstruktur abgelegt sind (Tabelle 3.2). Persistent gespeichert werden die Daten, indem man sie in XML serialisiert (vgl. Abschnitt 4.1).

Tabelle 3.2 Vollständige Jobbeschreibung

Feld-Name	Funktion	Bemerkung	Beispiel
jobID	Identifiziert einen Job eindeutig	wird vom Server verwaltet	0123456789987
jobType	Benutzer, dem der Job gehört und sein Projekt in der Form <user>_<project>, wird beim Verteilen der Jobs benutzt, dient der Bildung der Download-Pfade für die Jobdateien	muss vom Nutzer festgelegt werden	mbo_dissertation
status	Zustand des Jobs, konstante Werte: {FREE, WORKING, DONE, BLOCKED, AUTOBLOCKED}	wird vom Server verwaltet	DONE
countdown	Fehlertoleranz-Countdown des Jobs, konstante Werte: {5, 4, 3, 2, 1, 0}	wird vom Server verwaltet	3
platform	Zielplattform des Jobs in der Form <Software>.<Haupspeicher>.<Uploadmenge>.<CPU>; Software: Aneinanderreihung der Werte {W, L, N, M, J, P, Y, R, G}, diese stehen für Windows, Linux, .NET, Mono, Java, Perl, Python, R und GAMS. Läuft ein Job auf verschiedenen Plattformkombinationen, sind diese mit Semikolon zu trennen. Angaben für Hauptspeicher und Uploadmenge in Megabytes; CPU: Prozessorarchitektur, z. B. x86 oder AMD64	muss vom Nutzer festgelegt werden	WN;LMP.1024.*.* (= Windows mit .NET *ODER* Linux mit Mono und Perl, 1024 MB RAM notwendig, Uploaddatenmenge und CPU unspezifiziert)
command	Befehlszeile, die der Agent ausführen soll; dabei sind auch Scripte und Batchdateien (auch mit Ausgabeumleitung) möglich	muss vom Nutzer festgelegt werden	jaws.exe -l=10 -t=50
resultfiles	Namen der Ergebnisdateien, mehrere Dateien sind mit Semikolon zu trennen, darf auch „*“ sein, dann werden alle erzeugten Dateien zum Server geladen, Angabe von Unterverzeichnissen ist möglich	muss vom Nutzer festgelegt werden	*
savedresults	Namen der tatsächlich vom Agenten gespeicherten Ergebnisdateien, semikolongetrennt	wird vom Agenten erzeugt	result.data;errors\error.log

maintainoutput	Legt fest, ob die Ergebnisdateien Eingabedateien für andere Jobs sind, kann zur Reihenfolgesynchronisation genutzt werden, konstante Werte: {YES, NO}	muss vom Nutzer festgelegt werden	NO
files	Namen der Eingabedateien, semikolongetrennt, darf auch Wildcards wie „*“ enthalten, dann werden alle passenden Dateien heruntergeladen, Pfadangaben sind nicht möglich	muss vom Nutzer festgelegt werden	jaws.exe;*.dll
node	Name des ausführenden Agenten; besteht aus <Rechnername>[<Hardwarehash>]\<Benutzer-Account>.<Agenten-Prozess-ID>	Agent schickt Namen, Server ergänzt ggf. IP-Adresse	aifbacheron\[85fd8aa7] Bonn.1
time	Ausführungszeit des Jobs	wird vom Agenten gemessen	0:10:3.1976
mailnotification	Emailadresse für Benachrichtigung bei Job-Ende, kann leer bleiben	muss vom Nutzer festgelegt werden	mbo@aifb.uni-karlsruhe.de
nrzerocountdowns	Zählt mit, wie oft der Fehlertoleranz-Countdown dieses Jobs bereits abgelaufen ist	wird vom Server verwaltet	3
maxzerocountdowns	Legt fest, wie oft der Countdown ablaufen darf, bevor der Job den Status AUTOBLOCKED erhält	kann vom Nutzer geändert werden, default-Wert ist 5	2
maxrunningtime	Legt die maximale Laufzeit des Jobs fest, in Stunden (−1 heißt beliebig lange Laufzeit)	kann vom Nutzer geändert werden, default-Wert ist 24	48
rndID	Sicherungs-ID	wird vom Server verwaltet	4711
tempupload	Legt fest, ob die Ergebnisdateien des Jobs während der Laufzeit periodisch hochgeladen werden sollen, Werte sind {YES, NO}	muss vom Nutzer angegeben werden	NO
userIdentifier	Unter diesem Namen wird die Datei, die stdout, stderr und den Exit-Code enthält automatisch angelegt, darf auch leer bleiben	muss vom Nutzer angegeben werden	Test23.my1.eta12
userPriority	Priorität des Jobs innerhalb eines Benutzerkontextes (0: niedrigste, 9: höchste), wird beim Scheduling verwendet und bei AUTOBLOCKED-Jobs automatisch auf 0 gesetzt	kann vom Nutzer zur Laufzeit geändert werden, default-Wert ist 4	7
preUserIdentifiers	Enthält die userIdentifier derjenigen Jobs, die DONE sein müssen damit dieser Job gestartet werden kann, zur Reihenfolgemodellierung. Semikolongetrennte Liste, darf leer bleiben	muss vom Nutzer angegeben werden	Test23.my1.eta10; Test23.my1.eta11
commitTimestamp	Absoluter Zeitstempel bei DONE-Jobs, speichert Zeitpunkt des erfolgreichen Job-Abschlusses	wird vom Server verwaltet	633371888166868750

Die Attributwerte eines solchen Tupels werden grundsätzlich als Zeichenkette behandelt, gegebenenfalls (z. B. bei Rechenoperationen) werden temporär entsprechende Typumwandlungen durchgeführt.

Auf eine komplette Schnittstellenbeschreibung wird hier verzichtet. Exemplarisch soll an dieser Stelle nur die SOAP-Webmethode *GetJob()* beschrieben werden (Listing 3.1), mit der die Agenten beim Server nach Rechenjobs anfragen. Für die genaue Bedeutung und Syntax der einzelnen Parameter sei an dieser Stelle auf Abschnitt 3.4 verwiesen.

Listing 3.1 SOAP-Schnittstelle für Job-Anfragen

GetJob() Method

```
public string[] GetJob(
   string type, string agentPlatform,
   string node, string downloadLimit
);
```

Parameters

type
Gewünschter Jobtyp.

agentPlatform
Gibt an, welche Art von Jobs der Agent ausführen kann.

node
Damit muss der aufrufende Agent einen Namen mitschicken.

downloadLimit
Damit teilt der Agent mit, wie viele Daten er maximal downloaden will.

Return Value

Beschreibung des vom Agenten auszuführenden Jobs.

Fragt ein Agent beim Server nach einem Job, wird zunächst eine Liste mit allen Kandidaten gebildet, die für eine Ausführung auf dem entsprechenden Rechner in Frage kommen. Dabei führt der Server mit jedem Job der Datenbasis die folgenden Tests durch:

- Hat der Job den *status* FREE?
- Passt der vom Agent gewünschte *jobType* zu dem des Jobs? Hierbei wird geprüft, ob der gewünschte Typ einen Teilstring des Typs des Jobs darstellt.
- Entsprechen die Plattformanforderungen (*platform*) des Jobs dem, was der Agent anbietet bzw. wünscht (z. B. Betriebssystem oder Hauptspeicheranforderungen)?
- Sind alle in der Jobspezifikation angegeben *files* physisch vorhanden?
- Ist die Gesamtgröße dieser Dateien nicht größer als die vom Agent angegebene maximale Datenmenge?
- Sind alle eventuell angegebenen Vorgängerjobs (*preUserIdentifiers*) erledigt?

Wenn alle Tests positiv beantwortet werden, wird dieser Job in die Kandidatenliste aufgenommen, aus der dann vom Scheduler einer ausgewählt wird. Bevor jedoch dieses Auswahlverfahren beschrieben wird, soll zunächst erläutert werden, warum es überhaupt ein Problem darstellt, aus der Liste der Kandidaten einen geeigneten Job auszuwählen und warum man nicht einfach den erstbesten Job an den Client ausgibt.

3.3.3 Fehlertoleranz

Betreibt man einen dedizierten Rechencluster, kann man sich im Wesentlichen darauf verlassen, dass jeder Knoten einen Job vollständig bearbeitet, ohne vor Fertigstellung des Jobs auszufallen. Ein Rechner fällt normalerweise durch Hardware-Schäden, Fehler in der Stromversorgung oder durch Systemabstürze bzw. Softwarefehler aus, was jedoch beim Betrieb eines spezialisierten Rechenclusters vernachlässigbar ist. Betreibt man die Knoten

jedoch in einer unsicheren Umgebung, wie normalen Arbeitsplätzen oder in einem Studierendenpool, ist mit Ausfällen oder nicht vollständig durchgeführten Jobs zu rechnen. Die PCs können vom Benutzer einfach ausgeschaltet oder heruntergefahren werden, und die bisher auf diesem Knoten getätigten Berechnungen sind verloren. Aus diesem Grund wurden Mechanismen entwickelt, die diesem Problem entgegenwirken:

- Upload von Zwischenergebnissen vom Agenten zum Server
- Keep-Alive-Signal vom Agenten zum Server

Beim – vom Nutzer zu aktivierenden – Upload von Zwischenergebnissen überträgt der Agent in regelmäßigen Abständen alle bisher erzeugten Ergebnisdateien zum Server. So kann der Nutzer einer parallelen Anwendung entscheiden, ob der Job abgebrochen oder im Falle eines Ausfalls des Rechenknotens neu gestartet werden soll. Diese Entscheidung muss aber der Nutzer treffen und hängt natürlich auch von der Art des durch den Job bearbeiteten Problems ab. Blockiert man einen abgebrochenen Job nicht explizit, wird er nach einer gewissen Zeit neu gestartet: Der Server verwaltet für jeden gestarteten Job einen Zähler, der periodisch um 1 erniedrigt wird (siehe Abschnitt 3.3.2, Datenfeld *countdown*). Der Agent, auf dem der Job läuft, sendet periodisch ein Keep-Alive-Signal, das den Server veranlasst, den Zähler wieder auf den maximalen Initialwert zu setzen. Fällt der Agentenrechner aus, wird für diesen Job das regelmäßige Keep-Alive-Signal ausbleiben und der Zähler beim Server damit nach einer gewissen Zeit auf Null abgelaufen sein. In diesem Falle erhält der Job eine neue ID und ist für einen Neustart bereit. Da es durch Programmier- oder sonstige Softwarefehler der parallelen Anwendung vorkommen kann, dass ein Job auch auf zuverlässigen Rechenknoten immer abstürzt, existiert ein Mechanismus, der den Neustart eines Jobs nur endlich oft veranlasst: Jobs, die zu oft nicht erfolgreich bearbeitet werden, werden automatisch blockiert und können nur manuell vom Nutzer oder Administrator wieder zur Bearbeitung freigegeben werden. Die tolerierte Anzahl der Abstürze bis zur endgültigen Blockierung kann man selbst festlegen (siehe Abschnitt 3.3.2, Datenfeld *maxzerocountdowns*).

3.3.4 Monitoring der Rechenknoten

Beim Entwurf von JoSchKa wurde davon ausgegangen, dass sich die einzelnen Rechenknoten nicht nur in ihrer Hard- und Softwareausstattung unterscheiden, sondern auch im jeweiligen Zuverlässigkeitsverhalten: Es gibt Knoten, die einen Rechenjob unabhängig von dessen Laufzeit zuverlässig vollständig abarbeiten, ebenso existieren welche, die nicht vorhersagbar ausfallen. Weiterhin besteht die Möglichkeit, dass ein bisher sehr zuverlässiger Rechner sein Arbeitsverhalten ändert und unzuverlässig wird bzw. im umgekehrten Fall ein bisher unsicherer nun robust arbeitet.

Um die einzelnen Rechner beurteilen zu können, wurde eine Komponente entwickelt, die das Verhalten eines jeden am System bekannten Knotens beobachtet. Ziel ist es, über diese Messwerte Aussagen darüber treffen zu können, wie sich dieser in der näheren Zukunft verhalten wird und dieses Wissen bei der Verteilung der einzelnen Jobs berücksichtigen zu können [BoSc07].

Pro Knoten werden dabei die folgenden Werte protokolliert bzw. errechnet:

- B: Ein synthetischer Benchmarkindex, der die CPU-Leistung eines Rechenknotens repräsentiert, wobei $B \in \mathbb{R}$ und $B \in [-1,1]$ gilt.
- avF: Die durchschnittliche Arbeitszeit (gemessen in Echtzeit-Minuten), die der Knoten für einen nicht erfolgreich bearbeiteten Job bis zum Ausfall benötigt, wobei $avF \in \mathbb{N}$ gilt.
- avS: Die durchschnittliche Arbeitszeit (gemessen in Echtzeit-Minuten), die der Knoten für einen vollständig bearbeiteten Job benötigt, wobei $avS \in \mathbb{N}$ gilt.
- avU: Die durchschnittliche Betriebszeit (gemessen in Echtzeit-Minuten) des Knotens (also die Zeitspannen, in der der Rechner ohne Ausfall durchgängig verfügbar ist), wobei $avU \in \mathbb{N}$ gilt.
- R: Der Zuverlässigkeitsindex des Knotens, wobei $R \in \mathbb{R}$ und $R \in [-1,1]$ gilt.
- nP: Der normierte Leistungsindex des Knotens, der die Leistungsfähigkeit eines Knotens einer von $nPMAX+1$ verschiedenen Leistungsklassen zuteilt, wobei $nP \in \mathbb{N}$ und $nP \in \{0,\ldots,nPMAX\}$ gilt.

Im Folgenden bezeichne $EWA(v_n,\ldots,v_1)$ den exponentiell gewichteten Durchschnitt, bei dem ältere Messwerte weniger stark in die Berechnung eingehen als aktuelle, und der wie folgt rekursiv ermittelt wird: $EWA(v_n,\ldots,v_1) = \tilde{v}_n = \alpha v_n + (1-\alpha)\tilde{v}_{n-1}$ mit $\alpha \in \mathbb{R}$, $\alpha \in [0,1]$ und $\tilde{v}_1 = v_1$. Die v_i seien nach ihrem Alter sortiert, mit v_n als jüngstem Wert.

Formal berechnen sich die Werte für einen Knoten $NODE_i$, $i \in \mathbb{N}$, $i \in \{1,\ldots,l\}$, wie folgt:

- $$B_i = \begin{cases} 1, & rB_i < 5000 \\ ½, & 5000 \leq rB_i < 10000 \\ 0, & 10000 \leq rB_i < 15000 \\ -½, & 15000 \leq rB_i < 20000 \\ -1, & 20000 \leq rB_i \end{cases},$$

  wobei die $rB_i \in \mathbb{N}$ die in Millisekunden gemessene Echtzeit darstellen, die für die Durchführung eines fest definierten logisch-arithmetischen Benchmarks benötigt wird. Dieser Testlauf wird jeweils einmalig beim Start durchgeführt.
- $avF_i = EWA\left(f_{t_f},\ldots,f_1\right)$,

  wobei die f_j die letzten t_f Bearbeitungszeiten für nicht erfolgreich bearbeitete Jobs darstellen.
- $avS_i = EWA\left(s_{t_s},\ldots,s_1\right)$,

  wobei die s_j die letzten t_s Bearbeitungszeiten für erfolgreich bearbeitete Jobs darstellen.
- $avU_i = EWA\left(u_{t_u},\ldots,u_1\right)$,

  wobei die u_j die letzten t_u Betriebszeiten des Knotens darstellen.

- $R_i = EWA\left(r_{t_r},...,r_1\right)$,
  wobei die r_j jeweils Erfolgsmaß für die letzten t_r bearbeiteten Jobs darstellen, mit $r_j = 1$ für fertig gestellte und $r_j = -1$ für abgebrochene Jobs. R_i wird vom Server mit dem jeweiligen Benchmarkindex initialisiert, es gilt zunächst also $R_i = B_i$.
- $nP_i = \left\lfloor \frac{R_i - \min_j\left(R_j\right)}{\max_j\left(R_j\right) - \min_j\left(R_j\right)} \cdot nPMAX + 0{,}5 \right\rfloor$,
  hierbei wird also der Knoten mit dem kleinsten Zuverlässigkeitsindex der Leistungsklasse 0 zugeteilt, derjenige mit dem größten Zuverlässigkeitsindex wird der Klasse $nPMAX$ zugeteilt. Alle anderen werden entsprechend proportional den verbleibenden $nPMAX - 1$ Zwischenstufen zugeteilt.

Für die Praxis wurden $t_f = t_s = t_u = t_r = 10$, $\alpha = 0{,}25$, und $nPMAX = 20$ gesetzt. Die Werte(-bereiche) für B_i und $nPMAX$ wurden dabei so gewählt, um eine grobe Einteilung der Rechner in die zur Zeit gängigen Leistungsklassen von Prozessoren vornehmen zu können. Die Werte für B_i sind symmetrisch um 0 angelegt, um den anfangs noch unbekannten Zuverlässigkeitsindex R_i initialisieren zu können. Die in Abschnitt 3.3.6 durchgeführten Tests wurden mit verschiedenen Klassifizierungsgranularitäten und Parameterkombinationen durchgeführt, jedoch sei an dieser Stelle schon vorausgeschickt, dass sich andere als die hier festgelegten Werte nicht signifikant auf die Ergebnisse auswirken.

Diese Daten, die der Server für jeden einzelnen am System bekannten Rechner sammelt, lassen sich wie folgt zusammenfassen: Man weiß für jeden Knoten,

- wie schnell er einmalig einen definierten Benchmark abgearbeitet hat (*B*),
- wie lange er im Schnitt pro Job arbeitet, bevor er ausfällt (*avF*),
- wie lange er im Schnitt pro Job mindestens arbeitet, wenn er nicht ausfällt (*avS*),
- wie lange er im Schnitt verfügbar ist, bis er ausfällt (*avU*),
- wie zuverlässig er bei den letzten 10 bearbeiteten Jobs war (*R*) und
- wie zuverlässig er im Vergleich mit den anderen Knoten arbeitet (*nP*).

Dieses Wissen wird bei der leistungs-, der laufzeit- und bei der betriebszeitbasierten Verteilung der Jobs auf die einzelnen Rechenknoten verwendet. Die Verteilung und die dahinterliegenden Verfahren werden im folgenden Abschnitt beschrieben.

3.3.5 Verteilung der Jobs

Wie bereits deutlich wurde, hat man es in dem hier betrachteten Szenario nicht mit einem spezialisierten Rechencluster zu tun, sondern mit einem heterogenen Pool aus unterschiedlichsten Rechnern, die unterschiedliche Rechenleistung und vor allem unterschiedliche Zuverlässigkeit besitzen: Ein Rechner kann tagelang ununterbrochen laufen und dann plötzlich wiederholt unvorhersehbar neu gestartet werden. Ein Job, der eine Laufzeit von mehreren Stunden hat, kann auf einem derart unzuverlässigen Rechner nun nicht mehr ausgeführt werden. Gleichzeitig wäre es Verschwendung, wenn ein zuverlässiger

Knoten nur Jobs ausführt, die jeweils nach wenigen Minuten beendet sind. Wenn nun das Verteilsystem verschiedene Jobs verschiedener Nutzer zur Auswahl hat, liegt es nahe, sie so zu verteilen, dass die durch Ausfälle unnötig verwendete Rechenzeit möglichst klein gehalten wird, die einzelnen Nutzer jedoch so fair wie möglich behandelt werden, d. h. kein Nutzer soll das Gefühl haben, er würde benachteiligt. In diesem Abschnitt werden zunächst die grundlegenden Verteilstrategien von JoSchKa beschrieben, bevor sie dann bewertet und zu einem kombinierten Verteilalgorithmus zusammengeschaltet werden [BoSc07].

Insgesamt existieren fünf verschiedene Verteilverfahren, die im Anschluss ausführlich diskutiert werden:

(a) Ausbalancierte Verteilung: Hierbei werden die Jobs für alle Nutzer fair an die einzelnen Knoten verteilt, so dass für alle Nutzer die gleiche Anzahl davon arbeitet.

(b) Bevorzugung neuer Nutzer: Bei dieser Strategie werden die Jobs des Benutzers bevorzugt, von dem bisher die wenigsten Jobs berechnet wurden.

(c) Leistungsbasierte Verteilung: Zuverlässige Rechenknoten bearbeiten Jobs, die hohe Anforderungen an die Laufzeit haben, unzuverlässige Knoten bearbeiten Jobs mit geringen Anforderungen.

(d) Laufzeitbasierte Verteilung: Sie arbeitet ähnlich der leistungsbasierten Verteilung, jedoch basiert die Zuordnung von Jobs zu Rechnern nicht auf einem normierten Leistungsindex, sondern auf den realen Arbeitszeiten des Rechenknotens und den realen Laufzeitanforderungen des Jobs.

(e) Betriebszeitbasierte Verteilung: Sie entspricht im Wesentlichen der laufzeitbasierten Verteilung, arbeitet jedoch ausschließlich auf Basis der durchschnittlichen Verfügbarkeit der Knoten.

Bei den beschriebenen Verfahren unterscheidet der Scheduler normalerweise nicht zwischen einzelnen Benutzern, sondern zwischen einzelnen *JobType*s, (siehe auch Abschnitt 3.3.2), was den Vorteil hat, dass man auf möglicherweise unterschiedliche Anforderungen, die der gleiche Nutzer an das System stellt, besser eingehen kann. Sind weniger Agenten als *JobTypes* vorhanden, unterscheidet der Scheduler nur Benutzer. Die verschiedenen Jobtypen des gleichen Benutzers werden also gruppiert betrachtet, um auch in diesem Fall eine gerechte Verteilung ermöglichen zu können. Weiterhin wird eine vom Benutzer eventuell vorgegebene Priorisierung berücksichtigt. Unabhängig vom gewählten Verfahren werden die Jobs entsprechend ihrer Priorität ausgeliefert. Im Folgenden wird davon ausgegangen, dass im System l Knoten bekannt seien und n Jobs zur Verteilung bereitstehen, die sich auf m verschiedene Job-Typen bzw. Benutzer verteilen, wobei $n,m \in \mathbb{N}$ und $m \leq n$ gilt. Weiterhin bezeichne

- $J_i, i \in \{1,\ldots,n\}$ den Job i,
- $JT_k, k \in \{1,\ldots,m\}$ den Jobtyp k (bzw. den Benutzer k, falls $l < m$)
- $jT : \{1,\ldots,n\} \rightarrow \{1,\ldots,m\}$ mit $jT(i) = k$ der Jobtyp-Index des Jobs i.

Ausbalancierte Verteilung

Es sei $nrWORKING_k$ die Anzahl an Jobs des Typs JT_k, die zum Zeitpunkt der Anfrage des Knotens bearbeitet werden (wobei $nrWORKING_k \in \mathbb{N}$). Dann wird der Job i ausgeliefert, der $nrWORKING_{jT(i)}$ minimiert. Ein Knoten erhält also grundsätzlich einen Job des Typs, von dem gerade am wenigsten Jobs bearbeitet werden.

Dies hat zur Folge, dass zu jedem Zeitpunkt für jeden Typ/Benutzer die gleiche Anzahl an Rechenknoten aktiv ist. Dieses Gleichgewicht verschiebt sich erst dann, wenn nicht mehr genügend Jobs eines Nutzers zur Bearbeitung verfügbar sind.

Bevorzugung neuer Nutzer

Es sei $relDONE_k$ der Anteil an Jobs des Typs JT_k, die zum Zeitpunkt der Anfrage des Rechenknotens erfolgreich berechnet wurden (wobei $relDONE_k \in \mathbb{R}$ und $relDONE_k \in [0,1]$). Dann wird der Job i ausgeliefert, der $relDONE_{jT(i)}$ minimiert.

Dies hat zur Folge, dass Jobtypen bzw. Benutzer, deren relativer Anteil an abgeschlossenen Jobs am niedrigsten ist, alleinig bevorzugt werden, bis sich dieser Anteil an den der anderen Nutzer angeglichen hat.

Leistungsbasierte Verteilung

Bei dieser Verteilstrategie werden die vorhandenen Jobtypen zunächst nach ihren Laufzeitanforderungen klassifiziert. Dies geschieht analog zum normierten Leistungsindex eines Rechenknotens (siehe Abschnitt 3.3.4).

Dabei sei zunächst avT_k die (ebenfalls exponentiell gewichtet ermittelte) durchschnittliche Laufzeit eines Jobs vom Typ JT_k, gemessen in Minuten. Diese Laufzeit wird gemäß der folgenden Vorschrift auf einen synthetischen Laufzeitindex $avTI_k$ abgebildet:

$$avTI_k = \begin{cases} -1, & avT_k < 15 \\ -\tfrac{2}{3}, & 15 \le avT_k < 60 \\ -\tfrac{1}{3}, & 60 \le avT_k < 180 \\ 0, & 180 \le avT_k < 480 \\ \tfrac{1}{3}, & 480 \le avT_k < 960 \\ \tfrac{2}{3}, & 960 \le avT_k < 2160 \\ 1, & 2160 \le avT_k \end{cases}$$

Anhand dieses Laufzeitindex werden die einzelnen Typen in $TIMEMAX+1$ Klassen eingeteilt. Für die Laufzeitklasse $nTIME_k$ eines Jobs des Typs JT_k gilt: $nTIME_k \in \mathbb{N}$, $nTIME_k \in \{0,\ldots,TIMEMAX\}$ und sie berechnet sich wie folgt:

$$nTIME_k = \left\lfloor \frac{avTI_k - \min_j\left(avTI_j\right)}{\max_j\left(avTI_j\right) - \min_j\left(avTI_j\right)} \cdot TIMEMAX + 0{,}5 \right\rfloor$$

Analog zum Leistungsindex eines Knotens wurde hier ebenfalls $TIMEMAX = 20$ gewählt, um die gleiche Anzahl an Klassen zu erhalten. Wenn nun ein Knoten $NODE_j$ nach

einem Job anfragt, wird derjenige Job ausgeliefert, dessen Laufzeitklasse sich am wenigsten vom normierten Leistungsindex des Rechenknotens unterscheidet. Das ist also der Job J_i, der $\left| nTIME_{jT(i)} - nP_j \right|$ minimiert.

Die (Grenz-)Werte für die Einteilung der Jobs in die *avTI* -Klassen wurden anhand von typischen Laufzeiterfahrungen aus der Praxis gewählt, die Symmetrie der Werte um den Nullpunkt wurde in Analogie zum Benchmarkindex B_i eines Rechenknotens (vgl. Abschnitt 3.3.4) gewählt. Andere Einteilungsgrenzen wurden ebenfalls getestet, allerdings ebenfalls ohne signifikante Änderungen an den Testresultaten (siehe Abschnitt 3.3.6). Das Entscheidende bei diesem Verfahren ist die Möglichkeit, die Jobs anhand Ihrer Laufzeit und die Rechner anhand Ihrer Zuverlässigkeit in aufeinander abbildbare Klassen einteilen zu können. Das ist der Grund, warum $TIMEMAX = nPMAX = 20$ gesetzt wird. Diese beiden Werte sollten nicht zu groß und nicht zu klein gewählt werden, um ähnliche Jobs und ähnliche Rechner noch sinnvoll gruppieren zu können. Es wurden verschiedene Werte getestet, der Wert 20 hat sich als sinnvoll erwiesen. Das Verfahren soll im Folgenden durch ein Beispiel erläutert werden.

Angenommen, dem System wären 5 Rechenknoten $NODE_1,\ldots,NODE_5$ bekannt. Diese hätten die Zuverlässigkeitsindices:

$$R_1 = -0{,}8, \quad R_2 = -0{,}4, \quad R_3 = 0, \quad R_4 = 0{,}6, \quad R_5 = 1$$

Daraus würden sich (vgl. 3.3.4) die folgenden normierten Leistungsklassen ergeben:

$$nP_1 = 0, \quad nP_2 = 4, \quad nP_3 = 9, \quad nP_4 = 16, \quad nP_5 = 20$$

Weiterhin seien im System Jobs dreier verschiedener Jobtypen $JT_1,\ldots,JT_3$ vorhanden, die die folgenden Laufzeitanforderungen (in Minuten) hätten:

$$avT_1 = 5, \quad avT_2 = 40, \quad avT_3 = 190$$

Daraus würden sich dann die normierten Laufzeitklassen

$$nTIME_1 = 0, \quad nTIME_2 = 7 \quad nTIME_3 = 20$$

berechnen. Jobs vom Typ JT_1 würden also von $NODE_1$ berechnet werden, Jobs aus JT_2 würden an $NODE_2$ und an $NODE_3$ verteilt, $NODE_4$ und $NODE_5$ wären für JT_3 zuständig.

NODE	**1**				**2**					**3**							**4**				**5**
nP *nTIME*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
JT	**1**							**2**													**3**

Abb. 3.4 Beispiel zur leistungsbasierten Verteilung

Abb. 3.4 zeigt, wie die einzelnen Knoten und Jobtypen des Beispiels den einzelnen Klassen (in der Tabelle in der mittleren Zeile dargestellt) zugeordnet werden und welche Knoten für welche Typen zuständig sind. Das Verfahren sorgt also dafür, dass die zuverlässigen Knoten immer diejenigen Jobs bearbeiten, die die längste erwartete Laufzeit haben. Analog dazu bearbeiten die unzuverlässigen Knoten immer kurzlaufende Jobs. Sollte ein Knoten einer Klasse zugeteilt werden, die nicht eindeutig einer Jobtypen-Klasse zugeordnet werden kann (im Beispiel die Klassen 3 und 13), erfolgt die Zuteilung zufällig auf eine der beiden nächstliegenden Jobtypen-Klassen.

Laufzeitbasierte Verteilung

Der Algorithmus dieses Verfahrens berücksichtigt bei der Zuordnung eines Jobs zu einem Rechenknoten nicht nur dessen Zuverlässigkeit, sondern primär die durchschnittliche Zeit, die der Knoten mit dem Bearbeiten von Jobs beschäftigt war. Dabei wird unterschieden, ob der Rechner den Job erfolgreich abgeschlossen hat oder ob der Knoten ausgefallen ist, bevor der Job beendet wurde. Es werden also die im vorigen Abschnitt beschriebenen Werte avF_i und avS_i eines Rechenknotens verwendet. Wenn nun ein Knoten $NODE_i$ eine Anfrage an einen Job stellt, berechnet das Verfahren aus seinen Leistungsdaten zunächst einen Durchschnittszielwert $avTARGET$ wie folgt:

$$avTARGET = \begin{cases} \alpha \cdot avF_i, & R_i \in [-1,-\tfrac{2}{3}) \\ \beta \cdot avF_i, & R_i \in [-\tfrac{2}{3},-\tfrac{1}{3}) \\ \gamma \cdot avF_i, & R_i \in [-\tfrac{1}{3},0) \\ \delta \cdot avS_i, & R_i \in [0,\tfrac{1}{3}) \\ \varepsilon \cdot avS_i, & R_i \in [\tfrac{1}{3},\tfrac{2}{3}) \\ \zeta \cdot avS_i, & R_i \in [\tfrac{2}{3},1] \end{cases} \quad \text{mit} \quad \begin{array}{l} \alpha = 2^{-0,5s} \\ \beta = 2^{-0,25s} \\ \gamma = 1,0 \\ \delta = 1,0 \\ \varepsilon = 1,0 + 0,5s \\ \zeta = 1,0 + s \end{array} \quad \text{und } s \geq 0$$

Der Durchschnitts-Zielwert wird also aus den durchschnittlichen Arbeitszeiten des Knotens berechnet. In Abhängigkeit davon, ob der Rechner eher zuverlässig oder eher unzuverlässig ist, wird die durchschnittliche Erfolgszeit oder die durchschnittliche Fehlerzeit verwendet. Mit dem Parameter s kann der Durchschnittszielwert noch nach oben oder unten gespreizt werden, je nach dem Grad der (Un-)Zuverlässigkeit des Knotens. Der Grad der Spreizung kann dabei fest vorgegeben werden oder zur Laufzeit aus den durchschnittlichen Joblaufzeiten bestimmt werden. Kommt diese dynamische Spreizung zum Einsatz, berechnet sich der Steuerparameter s zur Laufzeit nach folgender Vorschrift:

$$s = \frac{\max_k \left(avT_k \right)}{\min_k \left(avT_k \right) \cdot m}$$

Die Spreizung soll es einem sehr zuverlässigen Knoten ermöglichen, einen $avTARGET$-Wert zu erhalten, der im Bereich des nächst laufzeitaufwändigeren Jobs liegt. Analog soll ein unzuverlässiger Knoten einen Job erhalten, dessen Laufzeit niedriger ist als der Knoten bisher im Schnitt arbeitsfähig war. Die mathematischen Formeln zur Berechnung der Faktoren $\alpha,\ldots,\zeta$ zur Spreizung der avF_i und avS_i sind so gewählt, dass die avF_i verkleinert, aber nicht kleiner als 0 werden. Die avS_i sollen dabei vergrößert werden, jedoch

nicht überproportional wachsen. Es wären an dieser Stelle durchaus andere Berechnungen (z. B. quadratische Funktionen) denkbar. Neben der Bestimmung der Spreizung verwendet das Verfahren die durchschnittlich zu erwartende Laufzeit avT_k der Jobtypen, um einen Laufzeitzielwert zu berechnen. Im Folgenden gelte $avT_i \leq avT_j$ (für $i < j$), sie seien also entsprechend ihrer Laufzeit sortiert. Für je zwei benachbarte Typen wird der Mittelwert der Laufzeiten bestimmt:

$$avTM_k = \frac{avT_k + avT_{k+1}}{2}$$

Schließlich wird der eigentliche Laufzeit-Zielwert $RLTV$ berechnet:

$$RLTV = RLTV^* + \mathrm{rnd}(-2,2), \text{ wobei}$$

$$RLTV^* = \begin{cases} avT^*, & \left|avTARGET - avT^*\right| < \left|avTARGET - avTM^*\right| \\ avTM^*, & \left|avTARGET - avT^*\right| \geq \left|avTARGET - avTM^*\right| \end{cases}, \text{ mit}$$

$$\left|avTARGET - avT^*\right| = \min_{k-1} \left|avTARGET - avT_k\right|, \text{ und}$$

$$\left|avTARGET - avTM^*\right| = \min_{k} \left|avTARGET - avTM_k\right|$$

Es wird also aus der Gesamtmenge der durchschnittlichen Joblaufzeiten und deren Mittelwerten derjenige Wert bestimmt, der dem zuverlässigkeitsabhängig gespreizten Durchschnittszielwert des anfragenden Knotens am nächsten liegt. Da das System hier nur mit Ganzzahlen rechnet, wird $RLTV$, um Rundungseffekte auszuschließen, noch um einen Wert von maximal 2 nach oben oder unten randomisiert. An den Knoten ausgeliefert wird letztlich der Job J_i, der $\left|RLTV - avT_{jT(i)}\right|$ minimiert. Die Verteilstrategie soll nun an einem Beispiel verdeutlicht werden.

Angenommen der Algorithmus sei mit dem Parameter $s = 2$ konfiguriert und in der Datenbasis seien Jobs der Typen $JT_1, \ldots, JT_4$ zur Bearbeitung verfügbar und diese hätten die folgenden durchschnittlichen Bearbeitungszeiten (in Minuten):

$$avT_1 = 20, \quad avT_2 = 50, \quad avT_3 = 150, \quad avT_4 = 160$$

Weiterhin sei angenommen, es würde nun ein Knoten am System nach einem Job anfragen und für diesen Knoten wären bisher die Leistungsdaten

$$avF = 60, \quad avS = 90, \quad R = 0{,}5$$

erfasst worden. Dann würde sich ein $avTARGET = avS \cdot (1{,}0 + 0{,}5s) = 180$ ergeben. Der diesem Wert am nächsten liegende Mittelwert ist $avTM_3 = \frac{1}{2}(avT_3 + avT_4) = 155$, die am nächsten liegende mittlere Joblaufzeit ist $avT_3 = 150$. Damit ergäbe sich $RLTV^* = 155$. Dem Knoten würde also wegen $RLTV = 155 + \mathrm{rnd}(-2,2)$ ein Job der Typen JT_3 oder JT_4 (zufällig ausgewählt) zugeteilt. Das Beispiel wird in Abb. 3.5 noch einmal graphisch veranschaulicht.

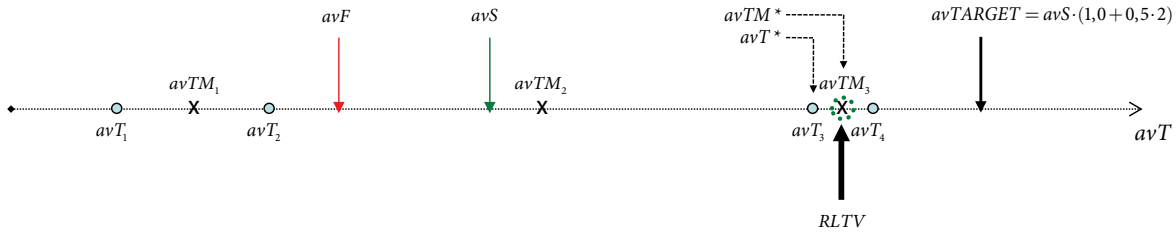


Abb. 3.5 Beispiel zur laufzeitbasierten Verteilung

Das Beispiel verdeutlicht auch, warum bei der Berechnung von $RLTV$ die größte durchschnittliche Joblaufzeit (avT_k, im Beispiel avT_4) ausgeschlossen wird: Die Rechner mit den längsten Laufzeiten würden andernfalls ausschließlich für den Nutzer mit den am längsten laufenden Jobs arbeiten. Zusätzlich wird klar, welche Funktion die Spreizung mit dem Parameter s besitzt: Ein Knoten, der sehr zuverlässig arbeitet, erhält dadurch die Chance, einen Job zu bearbeiten, der mehr Zeit beansprucht, als der Knoten bisher durchschnittlich gearbeitet hat.

Der Vollständigkeit halber werden noch die Intervallgrenzen, innerhalb derer der $avTARGET$-Wert eines Knotens liegen muss, um einem bestimmten Nutzer zugeteilt zu werden, angegeben. Es gelte im Folgenden:

- Ein Rechner erhält einen Job vom Typ JT_i, wenn $avTARGET$ im Intervall I_i liegt.
- Ein Rechner erhält einen Job vom Typ JT_i oder JT_{i+1} zufällig ausgewählt, wenn $avTARGET$ im Intervall $I_{i,i+1}$ liegt.

Die Intervallgrenzen des obigen Beispiels werden in der Abb. 3.6 nochmals verdeutlicht und ergeben sich für den allgemeinen Fall wie folgt:

- $I_i = \left[\frac{1}{2}\left(avTM_{i-1} + avT_i\right), \frac{1}{2}\left(avT_i + avTM_i\right)\right]$, mit $I_1 = \left[0, \frac{1}{2}\left(avT_i + avTM_i\right)\right]$
- $I_{i,i+1} = \left[\frac{1}{2}\left(avT_i + avTM_i\right), \frac{1}{2}\left(avTM_i + avT_{i+1}\right)\right]$, mit $I_{k-1,k} = \left[\left(avT_{k-1} + avTM_{k-1}\right), \infty\right)$

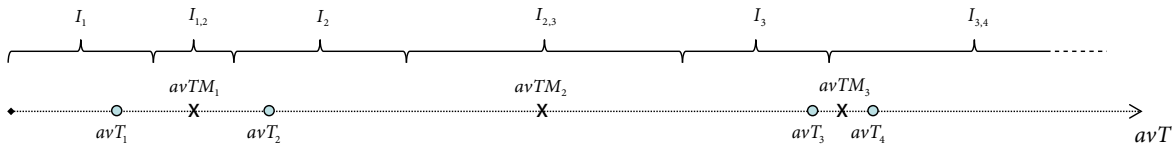


Abb. 3.6 Intervallgrenzen der Zuordnung von *avTARGET*-Werten zu Jobtypen

Betriebszeitbasierte Verteilung

Die betriebszeitbasierte Verteilstrategie berücksichtigt bei der Zuordnung von Jobs zu Rechenknoten die durchschnittliche Verfügbarkeit dieser Rechner, arbeitet aber ansonsten identisch zum laufzeitbasierten Verfahren. Der einzige Unterschied besteht in der Berechnung des $avTARGET$-Wertes:

$$avTARGET = \begin{cases} \alpha \cdot avU_i, & R_i \in [-1,-\frac{2}{3}) \\ \beta \cdot avU_i, & R_i \in [-\frac{2}{3},-\frac{1}{3}) \\ \gamma \cdot avU_i, & R_i \in [-\frac{1}{3},0) \\ \delta \cdot avU_i, & R_i \in [0,\frac{1}{3}) \\ \varepsilon \cdot avU_i, & R_i \in [\frac{1}{3},\frac{2}{3}) \\ \zeta \cdot avU_i, & R_i \in [\frac{2}{3},1] \end{cases} \quad \text{mit} \quad \begin{array}{l} \alpha = 2^{-0,5s} \\ \beta = 2^{-0,25s} \\ \gamma = 1,0 \\ \delta = 1,0 \\ \varepsilon = 1,0 + 0,5s \\ \zeta = 1,0 + s \end{array} \quad \text{und } s \geq 0$$

Unabhängig von der Zuverlässigkeit des Rechners wird immer die erfasste durchschnittliche Verfügbarkeitszeit avU verwendet, diese wird allerdings ebenfalls zuverlässigkeitsabhängig mit einem vorgegebenen oder zur Laufzeit berechneten Wert von s gespreizt. Für den Fall $s = 0$ gilt hier insbesondere: $avTARGET = avU_i$. Im weiteren Verlauf entspricht das Verfahren exakt dem laufzeitbasierten: Die durchschnittlichen Joblaufzeiten avT_i werden sortiert und die jeweiligen Mittelwerte $avTM_i$ werden bestimmt. Anschließend werden die $avTARGET$ am nächsten liegenden Werte avT^* und $avTM^*$ sowie aus diesen dann $RLTV$ bestimmt. An den anfragenden Knoten ausgeliefert wird wiederum der Job J_i, der $\left| RLTV - avT_{jT(i)} \right|$ minimiert.

Da das Verfahren im vorangegangenen Abschnitt ausführlich erläutert wurde, wird an dieser Stelle auf ein Beispiel verzichtet.

3.3.6 Simulation, Bewertung und Anwendung der Verteilstrategien

Um die einzelnen Strategien zu testen, zu bewerten und zu verbessern, wurde ein Simulator entwickelt, mit dessen Hilfe sich überprüfen lässt, wie sich das System bei bestimmten Jobtyp- und Rechnersituationen verhalten würde [BoSc07].

Für jeden zu simulierenden Rechenknoten lassen sich die folgenden Parameter konfigurieren:

- Den Benchmarkwert rB (in Millisekunden, siehe Abschnitt 3.3.4), den der Agent zu Beginn misst und zum Server überträgt,
- zwei Ausfallwahrscheinlichkeiten (in %), die festlegen, mit welcher Wahrscheinlichkeit ein Knoten pro Simulationsminute ausfällt und
- die Anzahl der Knoten, die auf diese Weise parametrisiert sind.

Durch die Angabe von zwei verschiedenen Ausfallwahrscheinlichkeiten kann simuliert werden, wie sich das System verhält, wenn ein Knoten plötzlich seine Zuverlässigkeit ändert. Nach 1000 simulierten Minuten wird bei jedem Knoten von der ersten auf die zweite Ausfallwahrscheinlichkeit umgeschaltet.

Jeder simulierte Jobtyp lässt sich wie folgt parametrisieren:

- Die Anzahl der Jobs dieses Typs und dessen Name,
- die Laufzeit (in Simulationsminuten), die ein Job dieses Typs benötigt und
- die Anzahl der Minuten, die nach Hinzufügen der Jobs dieses Typs simuliert werden sollen.

Die Konfiguration des Simulators erfolgt per XML und soll zunächst an einem Beispiel verdeutlicht werden (Listing 3.2):

Listing 3.2 Beispielhafte Simulationskonfiguration

```
<simConfig>
     <clients>
            <client cnt="10" power="4000" fail="0" fail2="2"/>
            <client cnt="15" power="12000" fail="1" fail2="0"/>
     </clients>
     <simulation>
            <step cnt="500" jobtype="job_L" jobduration="120" steps="50"/>
            <step cnt="6000" jobtype="job_S" jobduration="5" steps="2950"/>
     </simulation>
</simConfig>
```

Diese Konfiguration würde zunächst 500 Jobs des Typs *job_L* erzeugen und 50 Minuten simulieren, danach würden 6000 Jobs des Typs *job_S* mit anschließender Simulation von weiteren 2950 Minuten hinzugefügt werden. Ein *job_L*-Job würde nach 120 Minuten zu Ende sein und ein *job_S*-Job würde 5 Minuten brauchen. Das System würde 10 schnelle Rechner simulieren, die zunächst stabil arbeiten und nach 1000 Minuten häufig ausfallen. Weitere 15 langsame Knoten würden zunächst mäßig stabil arbeiten und nach 1000 Minuten zuverlässig werden.

Die im vorigen Abschnitt beschriebenen Verteilstrategien werden mit zwei Simulationskonfigurationen getestet, die sich nur in den simulierten Jobtypen, nicht aber bei den Rechenknoten unterscheiden. Die für beide Konfigurationen identische Rechnersituation ist die folgende (Listing 3.3):

Listing 3.3 Konfiguration der simulierten Rechenknoten

```
<clients>
     <client cnt="10" power="4000" fail="0" fail2="2"/>
     <client cnt="10" power="4000" fail="1" fail2="1"/>
     <client cnt="10" power="4000" fail="2" fail2="0"/>
     <client cnt="10" power="9000" fail="0" fail2="4"/>
     <client cnt="10" power="9000" fail="2" fail2="2"/>
     <client cnt="10" power="9000" fail="4" fail2="0"/>
     <client cnt="10" power="14000" fail="0" fail2="5"/>
     <client cnt="10" power="14000" fail="3" fail2="3"/>
     <client cnt="10" power="14000" fail="5" fail2="0"/>
     <client cnt="10" power="19000" fail="0" fail2="6"/>
     <client cnt="10" power="19000" fail="3" fail2="3"/>
     <client cnt="10" power="19000" fail="6" fail2="0"/>
</clients>
```

Es handelt sich um ein heterogenes Feld von Rechenknoten unterschiedlichster Leistung und Zuverlässigkeit. Die Jobs der Simulation A sind entsprechend Listing 3.4 definiert.

Listing 3.4 Jobkonfiguration Simulation A

```
<simulation> <!--Simulation A-->
     <step cnt="500" jobtype="long" jobduration="120" steps="50"/>
     <step cnt="1000" jobtype="medium" jobduration="35" steps="50"/>
     <step cnt="6000" jobtype="short" jobduration="5" steps="2890"/>
</simulation>
```

Die drei Typen unterscheiden sich in ihrer Laufzeit also recht deutlich voneinander. Im Gegensatz dazu verhalten sich bei Simulation B (Listing 3.5) zwei der drei Typen sehr ähnlich:

Listing 3.5 Jobkonfiguration Simulation B

```
<simulation> <!--Simulation B-->
     <step cnt="300" jobtype="long1" jobduration="120" steps="5"/>
     <step cnt="300" jobtype="long2" jobduration="130" steps="5"/>
     <step cnt="6000" jobtype="short" jobduration="10" steps="2990"/>
</simulation>
```

Um die Leistung der Verteilstrategien quantifizieren zu können, werden zwei Bewertungskriterien definiert:

(a) Die *durchschnittliche Effizienz avEff* (angegeben in Prozent) einer Strategie berechnet sich aus dem Verhältnis von erfolgreich gearbeiteten Simulationsminuten aller Knoten zur Gesamtzahl der von allen Knoten gearbeiteten Zeit. Gemessen und gemittelt wird dies über den gesamten Simulationszeitraum. Ein hoher Wert bedeutet hier, dass die einzelnen Knoten nicht sinnlos gearbeitet haben (was z. B. bei häufigen Abstürzen der Fall wäre).

(b) Die *gesamtdurchschnittliche Fertigungsrate avDONE* (ebenfalls in Prozent angegeben) berechnet sich aus dem Mittelwert der durchschnittlichen Fertigungsraten der einzelnen Jobtypen. Gemessen und gemittelt wird dies ebenfalls über den gesamten Simulationszeitraum. Ein hoher Wert würde hier bedeuten, dass die einzelnen Jobs schnell erledigt werden.

Im Folgenden werden die fünf Verteilverfahren mit den beiden Simulationen getestet und bewertet. Beim laufzeitbasierten Verfahren wird zusätzlich noch auf die Rolle des Spreizparameters *s* eingegangen. Die wesentlichen Fragestellungen bei der Beurteilung der einzelnen Verfahren lauten:

- Sind die Werte *avEff* und *avDONE* möglichst hoch?
- Sind die verschiedenen Jobtypen gut ausbalanciert?
- Wie adaptiv verhält sich die Verteilstrategie, wenn die Rechenknoten nach der 1000. Simulationsminute ihre Zuverlässigkeit ändern?

Zu jeder Verteilstrategie werden dazu zwei Diagramme (jeweils eines für Simulation A und eines für Simulation B) und eine Tabelle angegeben. Die jeweiligen Diagramme zeigen für jeden simulierten Jobtyp zwei farbgleiche Kurven, die angeben, wie sich die Fertigungsraten (%DONE) und die Anzahl der für einen Typ arbeitenden Knoten (#WOR-

KING) im Verlauf der Simulation entwickeln. Die schwarze gepunktete Effizienzkurve stellt das Verhältnis von erfolgreich genutzter Rechenzeit zur Gesamtrechenzeit über alle Knoten im Simulationsverlauf dar. Die direkt unterhalb der Diagramme angegebenen Tabellen fassen die Bewertung der simulierten Verteilstrategie kurz zusammen.

Ausbalancierte Verteilung

Charakteristisches Merkmal des ausbalancierten Verfahrens ist, dass für alle Jobtypen immer die gleiche Anzahl an Knoten tätig ist (Abb. 3.7): Die jeweiligen #WORKING-Kurven verlaufen alle auf der gleichen Höhe. Jeder der drei Jobtypen wird also zunächst durchweg von einem Drittel der Knoten bedient.

Ist ein Jobtyp vollständig bearbeitet, verteilen sich die Knoten zu jeweils einer Hälfte auf die verbleibenden beiden Typen, was im linken Teil der Abb. 3.7 gut zu sehen ist.

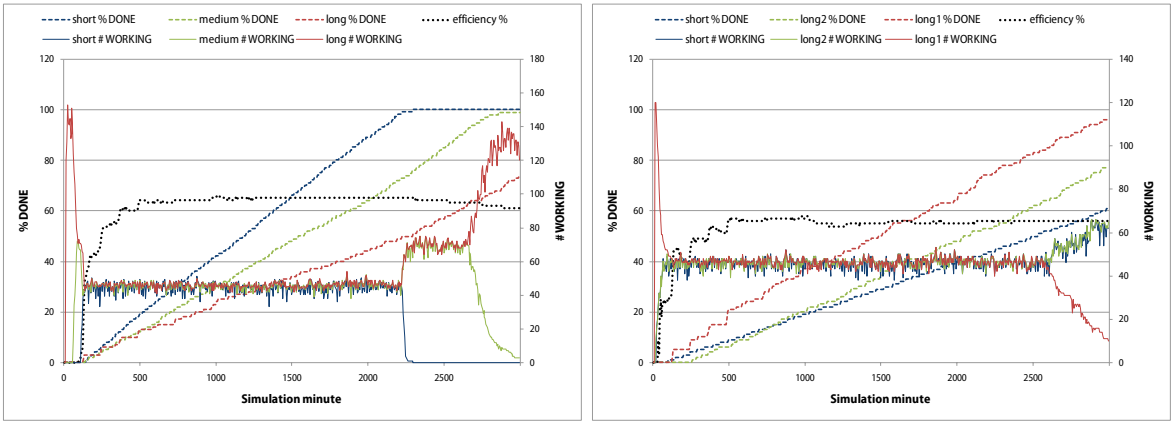


Abb. 3.7 Simulationen A / B bei ausbalancierter Verteilung

Tabelle 3.3 Resultate bei ausbalancierter Verteilung

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	60%	48%	sehr gut ausbalanciert / keine Reaktion auf Zuverlässigkeitsänderung, sehr stabil
B	53%	38%	sehr gut ausbalanciert / keine Reaktion auf Zuverlässigkeitsänderung, sehr stabil

Da die Zuverlässigkeit der Knoten nicht berücksichtig wird, hat die Änderung der Ausfallwahrscheinlichkeit keinen Einfluss auf die Verteilung. Bei diesem Verfahren kann und wird es passieren, dass ein sehr zuverlässiger Rechenknoten sehr kurz laufende Jobs bearbeitet, während unzuverlässige Knoten lange laufende Jobs berechnen müssen (die sie dann mit hoher Wahrscheinlichkeit nicht zu Ende bringen). Diese Strategie ist die algorithmisch am einfachsten zu realisierende und sie garantiert überdies eine gerechte Verteilung der Knoten an die einzelnen Benutzer. Aus diesem Grund ist die ausbalancierte Verteilung und ihre gemessenen Leistungsdaten (Tabelle 3.3) das Verfahren, das es zu verbessern gilt.

Bevorzugung neuer Nutzer

Diese Strategie bevorzugt immer denjenigen Jobtypen, der die niedrigste Fertigungsrate besitzt. Die logische Konsequenz ist, dass die Anzahl der für einen Typ arbeitenden Knoten (#WORKING) solange maximal ist, bis die Fertigungsrate die Rate der anderen Benutzer erreicht oder überholt hat, danach fällt sie auf 0 zurück (Abb. 3.8). Da die Zuverlässigkeit der Knoten bei dieser Strategie ebenfalls nicht berücksichtig wird, hat die Änderung der Ausfallwahrscheinlichkeit keinen Einfluss auf die Verteilung. Gegenüber der ausbalancierten Verteilung sinken vor allem bei Simulation A die Effizienz und der Fertigungsratendurchschnitt stark ab (Tabelle 3.4).

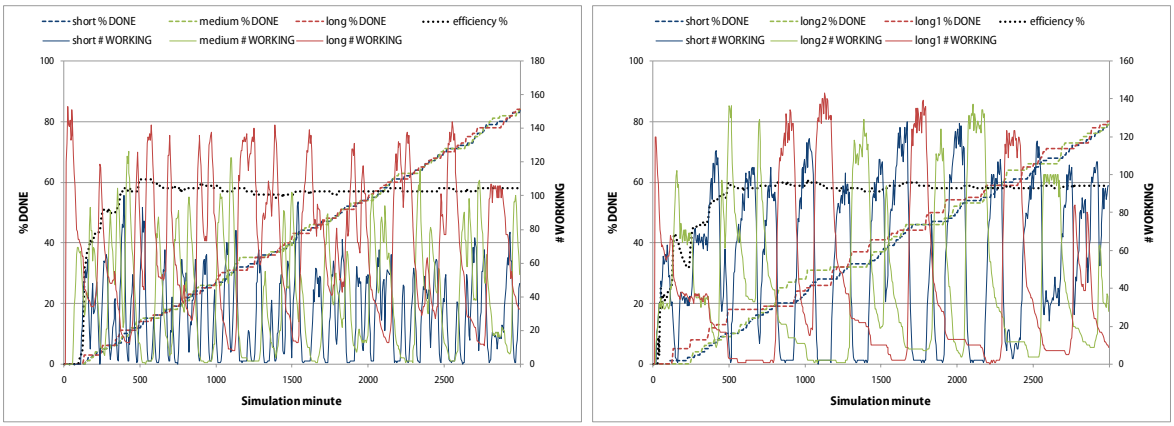


Abb. 3.8 Simulationen A / B bei Bevorzugung neuer Nutzer

Tabelle 3.4 Resultate bei Bevorzugung neuer Nutzer

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	54%	41%	Balance schwankt extrem / keine Reaktion auf Zuverlässigkeitsänderung, sehr stabil
B	55%	38%	Balance schwankt extrem / keine Reaktion auf Zuverlässigkeitsänderung, sehr stabil

Leistungsbasierte Verteilung

Auffällig ist bei diesem Verfahren zunächst, dass vor allem bei Simulation A die erzielte Effizienz und die Fertigungsraten nicht an die Werte der strikt ausbalancierten Verteilung heranreichen (Tabelle 3.5). Da dieses Verfahren die Jobs aufgrund der Zuverlässigkeit der Knoten verteilt, können nicht alle Typen gleich fair behandelt werden. Offensichtlich passen die Zuverlässigkeitswerte der meisten Knoten bei beiden Simulationen (Abb. 3.9) zunächst am besten zu den Anforderungen der mittellang laufenden Jobs. Weiterhin fällt auf, dass die Anzahl der für einen Typ arbeitenden Knoten wechselt, was daran liegt, dass ein Knoten seinen Zuverlässigkeitsindex durch Bearbeitung kurz laufender Jobs steigert und danach länger laufende Jobs zugeteilt bekommt. Kann er diese nicht erfolgreich abarbeiten, sinkt der Index und als Folge davon erhält er wieder kürzer laufende Jobs. Dieses Hin- und Herwandern der Knoten zwischen den einzelnen Leistungsklassen kann man gut an den *#WORKING*-Kurven der beiden Diagramme ablesen: Ihre Werte schwanken im Verlauf der Simulation.

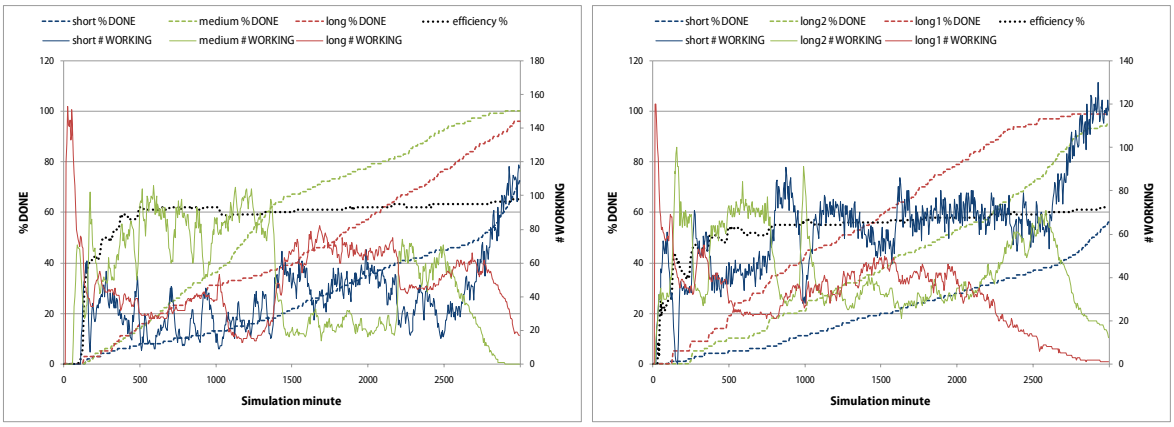


Abb. 3.9 Simulationen A / B bei leistungsbasierter Verteilung

Tabelle 3.5 Resultate bei leistungsbasierter Verteilung

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	58%	42%	gut ausbalanciert, mit Schwankungen / berücksichtigt Zuverlässigkeitsänderung, stabiles Gleichgewicht wird wieder hergestellt
B	54%	39%	gut ausbalanciert, mit Schwankungen / berücksichtigt Zuverlässigkeitsänderung, stabiles Gleichgewicht wird wieder hergestellt

Da das Verfahren ausschließlich die Zuverlässigkeit der Rechner zum Verteilen berücksichtigt, reagiert es gut auf die Zuverlässigkeitsänderungen: Das vor der Änderung herrschende relativ gut ausbalancierte Gleichgewicht stellt sich nach kurzer Zeit wieder ein.

Laufzeitbasierte Verteilung

Die laufzeitbasierte Verteilung teilt die Jobs den Knoten aufgrund deren Arbeitszeitverhalten zu. Ein Knoten, der eine bestimmte Zeit zuverlässig arbeiten kann, ohne auszufallen, wird Jobs bearbeiten, die entsprechende Laufzeitanforderungen haben.

Das erklärt die vor allem im ersten Drittel der Simulation A hohen Effizienzwerte von 70%–80% (Tabelle 3.6). Das Verfahren schafft es aber nicht, angemessen auf eine veränderte Ausfallwahrscheinlichkeit zu reagieren. Ein unzuverlässiger Knoten, der zuverlässig wird, erhält nicht die jetzt angemessenen langlaufenden Jobs. Ein solcher Knoten kann erst dann einen lang laufenden Job erhalten, wenn sein Arbeitszeitdurchschnitt angewachsen ist. Der wächst aber eben nur dann an, wenn lange Jobs erfolgreich bearbeitet werden. Berücksichtigt man also die Zuverlässigkeit eines Knotens nicht, sondern verwendet ausschließlich seine durchschnittlichen Erfolgs- oder Fehlerzeiten, wird das Verfahren instabil. Positive Zuverlässigkeitsänderungen werden nicht berücksichtigt.

Den Effekt sieht man vor allem im rechten Teil der Abb. 3.10: Nach der Zuverlässigkeitsänderung im Simulationsschritt 1000 werden die beiden lange laufenden Jobtypen nicht mehr berücksichtigt. Den Knoten, die zuverlässig waren und jetzt verstärkt ausfallen, werden richtigerweise die kurz laufenden Jobs zugeteilt, die Rechner jedoch, die unzuverlässig waren und jetzt stabil arbeiten, werden nicht den lange laufenden Typen zugeteilt.

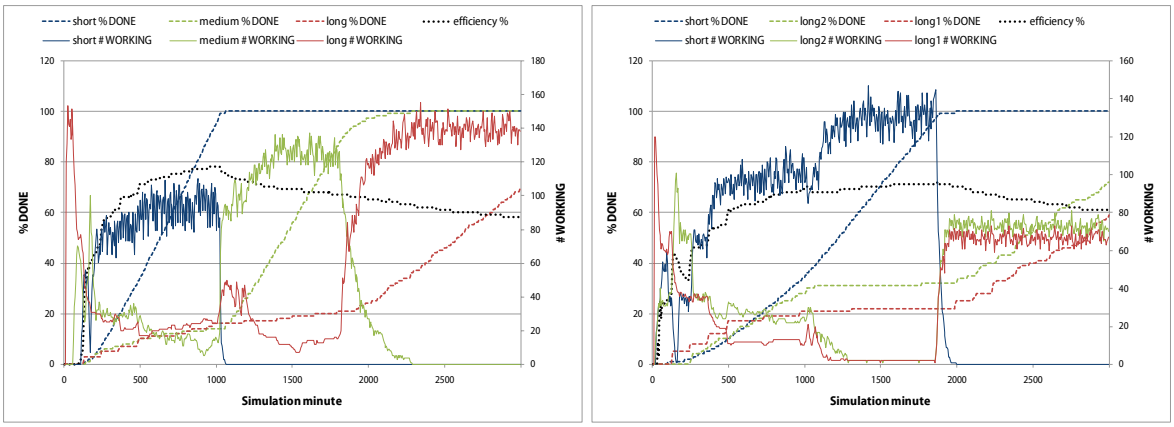


Abb. 3.10 Simulationen A / B bei laufzeitbasierter Verteilung mit $s = 0$

Tabelle 3.6 Resultate bei laufzeitbasierter Verteilung mit $s = 0$

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	63%	52%	Balance entsprechend dem Laufzeitverhalten / instabil, vorhergehendes Gleichgewicht stellt sich nicht wieder ein
B	62%	39%	Balance entsprechend dem Laufzeitverhalten / sehr instabil, vorhergehendes Gleichgewicht stellt sich nicht wieder ein, ignoriert langlaufende Job-Typen

Das Verfahren muss also so geändert werden, dass zuverlässige Rechner die Chance erhalten, Jobs zu berechnen, die mehr Zeit benötigen, als der Knoten bisher im Schnitt erfolgreich gearbeitet hat. Ein Knoten, der z. B. durchschnittlich 30 Minuten pro Job benötigt, um ihn erfolgreich abzuschließen, und dies zuverlässig tut, sollte ab und zu die Chance erhalten, auch einen 60 Minuten laufenden Job zu bearbeiten. Diese Spreizung, die durch den im vorigen Abschnitt vorgestellten Parameter *s* gesteuert wird, soll es einem Knoten erleichtern, zwischen den verschiedenen Jobtypen hin- und herzuwandern, wenn sich seine Zuverlässigkeit ändert, so wie das bei der leistungsbasierten Verteilung der Fall ist. Das laufzeitbasierte Verfahren wurde auf beiden Simulationen mit mehreren Einstellungen des Spreizungsparameters getestet, als Beispiel sei hier die Variante mit $s = 2$ gezeigt (Abb. 3.11).

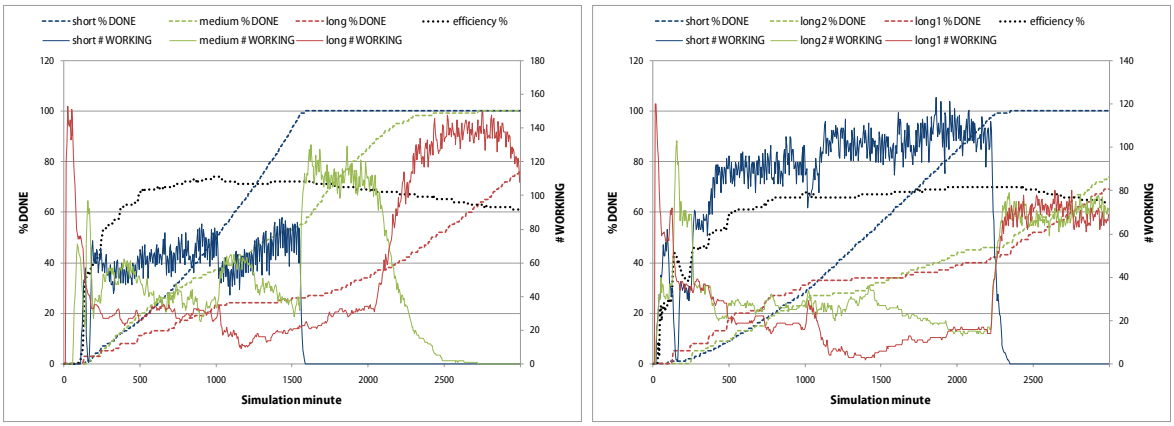


Abb. 3.11 Simulationen A / B bei laufzeitbasierter Verteilung mit $s = 2$

Tabelle 3.7 Resultate bei laufzeitbasierter Verteilung mit *s* = 2

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	64%	52%	Balance entsprechend dem Laufzeitverhalten / vorhergehendes Gleichgewicht stellt sich nach einiger Zeit wieder ein
B	62%	41%	Balance entsprechend dem Laufzeitverhalten / instabil, vorhergehendes Gleichgewicht stellt sich nicht wieder ein, ignoriert langlaufende Jobtypen, jedoch verbessert gegenüber s = 0

Die jetzt erzielten Fertigungsraten ähneln denen mit $s = 0$ und die Effizienzwerte sind nicht mehr ganz so hoch (Tabelle 3.7). Das Verfahren verhält sich nun aber bei beiden Simulationen stabiler. Leider wird bei der Simulation B der erste lang laufende Typ nach dem Umschalten der Ausfallwahrscheinlichkeiten immer noch benachteiligt, obwohl seine Anforderungen denen des zweiten lang laufenden Typs sehr ähneln.

Gibt man den Spreizwert nicht fest vor, sondern lässt ihn zur Laufzeit dynamisch berechnen, zeigt sich vor allem für die Simulation B ein ähnliches Bild (Abb. 3.12). Offensichtlich berechnet sich bei der Simulation B der Spreizparameter zu Werten, die recht nahe an $s = 2$ liegen. Bei der Simulation A sieht es jedoch anders aus: Gegenüber $s = 2$ ist die gesamte Verteilung ausgeglichener, auf die Zuverlässigkeitsänderung reagiert das Verfahren hier sehr gut, die gute Ausbalancierung stellt sich schnell wieder ein, wobei die Fertigungsraten der langlaufenden Jobs konstant weiterwachsen. Die Effizienz und die Gesamtfertigungsraten sind ähnlich gut geblieben (Tabelle 3.8).

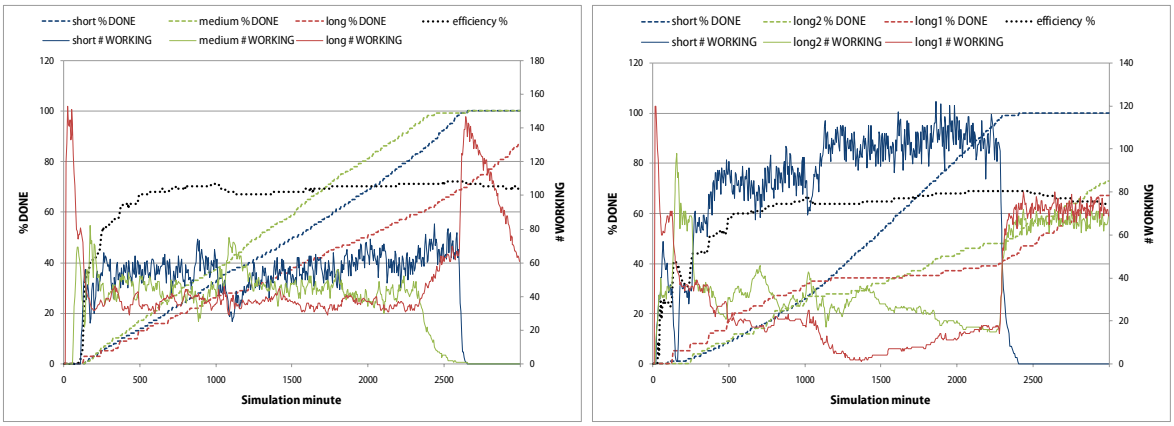


Abb. 3.12 Simulationen A / B bei laufzeitbasierter Verteilung mit *s* dynamisch

Tabelle 3.8 Resultate bei laufzeitbasierter Verteilung mit *s* dynamisch

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	65%	49%	Balance entsprechend dem Laufzeitverhalten / vorhergehendes Gleichgewicht stellt sich sehr schnell wieder ein
B	61%	40%	Balance entsprechend dem Laufzeitverhalten / instabil, vorhergehendes Gleichgewicht stellt sich nicht wieder ein, ignoriert langlaufende Jobtypen, jedoch verbessert gegenüber s = 0

Betriebszeitbasierte Verteilung

Im Gegensatz zur laufzeitbasierten Verteilung werden hier die Jobs entsprechend den Verfügbarkeitszeiten der einzelnen Knoten an diese verteilt. Je länger ein Rechner läuft, desto länger dürfen die Jobs laufen, die an ihn verteilt werden. Zu erwarten ist hier, dass sich das Verfahren bei Zuverlässigkeitsänderungen stabiler als das laufzeitbasierte verhält. Da der durchschnittliche Verfügbarkeitswert eines zuverlässig gewordenen Knotens automatisch steigt – und zwar unabhängig davon, welche Jobs er bearbeitet – müsste sich das Verfahren nach dem Kippen der Zuverlässigkeitswerte auch ohne Spreizung selbst wieder stabilisieren. Die Simulation mit $s = 0$ zeigt, dass diese Prognose tatsächlich eintritt (Abb. 3.13): Das Gleichgewicht der #WORKING-Kurven stellt sich nach dem Kippen der Zuverlässigkeitswerte schnell wieder ein.

Die Fertigungsraten sowie die Effizienzwerte (Tabelle 3.9) entsprechen denen der laufzeitbasierten Verteilung, das Verfahren reagiert aber selbst ohne Spreizung wesentlich stabiler auf Zuverlässigkeitsänderungen in der Simulation B: Während beim laufzeitbasierten Verfahren die beiden lang laufenden Jobs nach dem Kippen der Zuverlässigkeit nicht mehr berücksichtigt wurden, steigt die Verfügbarkeit der jetzt zuverlässig gewordenen Knoten von selbst an, worauf der Scheduler sie nach kurzer Zeit wieder den lang laufenden Jobs zuteilen kann.

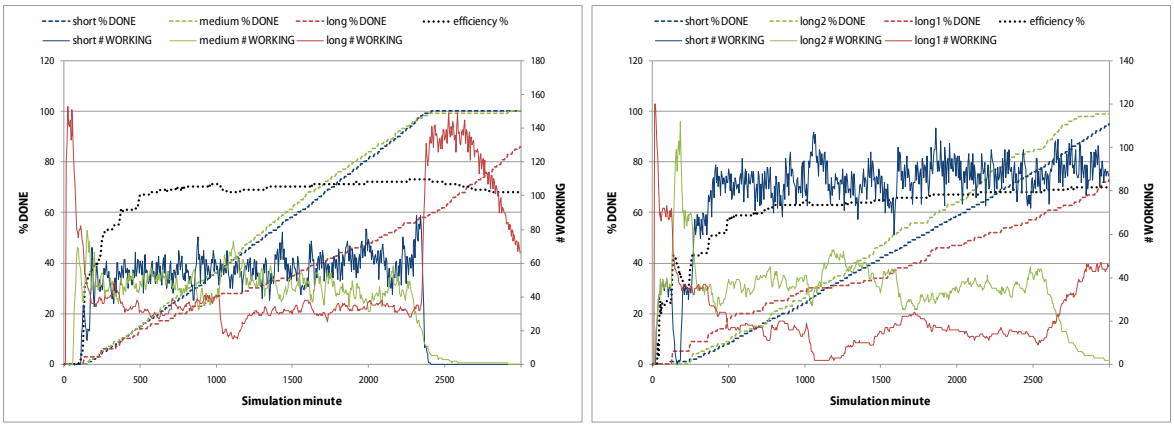


Abb. 3.13 Simulationen A / B bei betriebszeitbasierter Verteilung mit $s = 0$

Tabelle 3.9 Resultate bei betriebszeitbasierter Verteilung mit $s = 0$

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	65%	51%	Balance entsprechend dem Laufzeitverhalten / vorhergehendes Gleichgewicht stellt sich sofort wieder ein
B	61%	42%	Balance entsprechend dem Laufzeitverhalten / vorhergehendes Gleichgewicht stellt sich recht schnell wieder ein

Dass das Verfahren prinzipiell sogar völlig ohne die Spreizung auskommt, zeigt die folgende Simulation (Abb. 3.14), bei der *s* dynamisch berechnet wird: Das Kippen der Zuverlässigkeitswerte stört das Gleichgewicht der #WORKING-Kurven praktisch nicht.

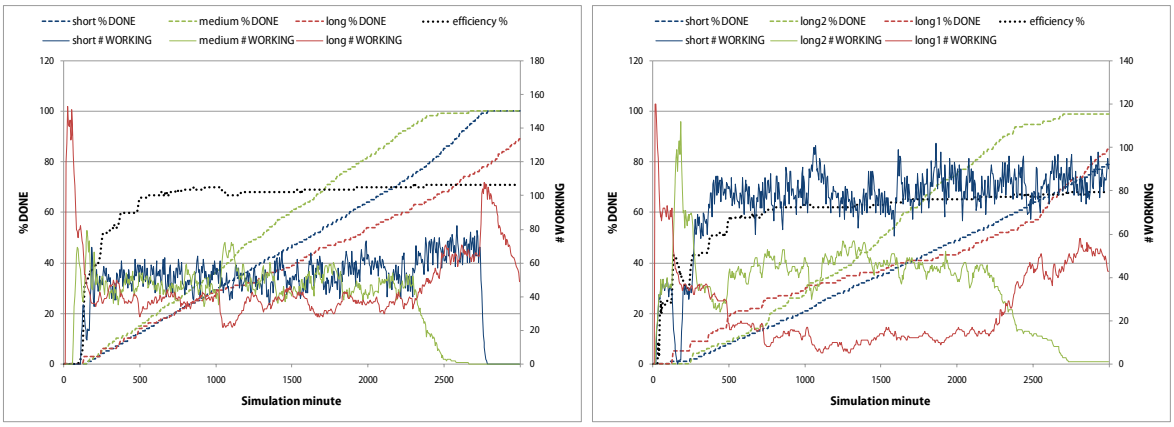


Abb. 3.14 Simulationen A / B bei betriebszeitbasierter Verteilung mit *s* dynamisch

Tabelle 3.10 Resultate bei betriebszeitbasierter Verteilung mit *s* dynamisch

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	64%	49%	Balance entsprechend dem Laufzeitverhalten / vorhergehendes Gleichgewicht stellt sich sofort wieder ein
B	60%	42%	Balance entsprechend dem Laufzeitverhalten / vorhergehendes Gleichgewicht stellt sich sofort wieder ein

Die Effizienzwerte und (bei Simulation A) die Fertigungsraten sind geringfügig niedriger (Tabelle 3.10) als ohne Spreizung, wohingegen sich das Verfahren jetzt noch schneller wieder stabilisiert und sich die Menge der für einen Nutzer arbeitenden Knoten auf das Niveau vor der Zuverlässigkeitsänderung einpendelt.

Im Folgenden wurden Überlegungen angestellt, wie sich die vorgestellten Verfahren zu einem Gesamtalgorithmus kombinieren lassen, der hohe Effizienz und Fertigungsraten erzielt, stabil gegenüber Zuverlässigkeitsänderung ist und die einzelnen Nutzer – entsprechend ihren Laufzeitanforderungen – trotzdem gerecht behandelt. Nutzer mit gleichen Anforderungen sollten die gleiche Anzahl an Rechenknoten bzw. die gleiche Rechenleistung erhalten.

Kombiniertes Verteilungsverfahren

Das kombinierte Verfahren vereint die Vorteile der fünf einzelnen Verfahren, die jeweiligen Nachteile werden kompensiert. Ideal wäre nämlich ein Verfahren, das die hohen Effizienzwerte und Fertigungsraten des laufzeitbasierten bzw. des betriebszeitbasierten Verfahrens erreicht, dabei auf Zuverlässigkeitsänderungen so robust und adaptiv wie das leistungsbasierte Verfahren reagiert und gleichzeitig die Verteilung der Knoten auf die einzelnen Jobtypen so ausbalanciert gestaltet, dass jeder Nutzer ausreichend fair behandelt wird. Dieses bezüglich mehrerer Kriterien günstige Verhalten sollte bei allen möglichen Kombinationen aus Rechenknoten (und deren Leistungs- bzw. Zuverlässigkeitscharakteristik) und Jobanforderungen erreicht werden.

Das resultierende Verfahren wird von fünf Parametern gesteuert:

- $fairLevel, \in \mathbb{R}, \in [0,1]$: Der Parameter steuert den Grad der fairen/ausbalancierten Verteilung der Knoten auf die Nutzer, der mindestens erfüllt sein muss.
- $doneRateLowBoost, \in \mathbb{R}, \in [0,1]$: Dieser Wert legt fest, bis zu welcher Grenze Jobtypen bevorzugt werden, die einen niedrigen Fertigungsgrad haben.
- $powerIndexProb, \in \mathbb{R}, \in [0,1]$: Die Wahrscheinlichkeit, mit der das leistungsbasierte Verteilverfahren anstelle des laufzeitbasierten Verfahrens eingesetzt wird.
- $runlengthScale, \in \mathbb{R}_0^+$: Der aus dem vorangegangenen Abschnitt bekannte Parameter *s* des laufzeit- und des betriebszeitbasierten Verfahrens.
- $useUptimes, \in \{\text{true}, \text{false}\}$: Der Boolesche Wert steuert, ob das betriebszeitbasierte Verfahren gegenüber dem laufzeitbasierten verwendet wird.

Wenn nun ein Rechenknoten $NODE_i$ beim Server nach einem Job anfragt, läuft der folgende Algorithmus ab:

(1) Zunächst bestimmt der Server alle Jobkandidaten, die für eine Zuteilung zu diesem Knoten in Frage kommen (siehe Abschnitt 3.3.2). Gleichzeitig werden für jeden am System bekannten Job-Typ JT_k die Werte $nrWORKING_k$, $relDONE_k$, und $avTI_k$ bestimmt.

(2) Anschließend wird ein Sonderfall behandelt: Die einzelnen Agenten können in einem Bildschirmschoner-Modus betrieben werden (vgl. Abschnitt 4.2). Da man davon ausgehen kann, dass ein Bildschirmschoner nur für sehr kurze Zeit aktiv ist, wird an Bildschirmschoner-Agenten immer der Job i ausgeliefert, der $avTI_{jT(i)}$ minimiert und der Algorithmus beendet.

(3) Die Zuverlässigkeitswerte R_i aller bekannten Knoten werden sortiert, aus den sortierten Werten wird $majIntvl = Q_{0,9} - Q_{0,1}$ (also die Breite des Intervalls, in dem die mittleren 80% der Zuverlässigkeitswerte liegen) bestimmt. Weiterhin wird der Wert $avTIdiff = \max_k(avTI_k) - \min_k(avTI_k)$ berechnet.

(4) Mit diesen Werten wird in manchen Fällen der (administrativ vorgegebene) Grad der Ausbalancierung erhöht:

- $(avTIdiff < 0,5 \vee majIntvl < 0,4) \wedge fairLevel < 0,33 \Rightarrow fairLevel = 0,33$
- $(avTIdiff = 0 \vee majIntvl < 0,2) \wedge fairLevel < 0,67 \Rightarrow fairLevel = 0,67$

Sind die Anforderungen der einzelnen Jobs zu ähnlich oder verhalten sich die am System bekannten Rechenknoten überwiegend gleich zuverlässig, braucht man die aufwändigen Leistungs- und Laufzeitverfahren nicht anzuwenden. Es ist in diesem Fall besser, durch verstärkte Anwendung des ausbalancierenden Verfahrens die Jobs fair auf die einzelnen Knoten zu verteilen.

(5) Falls $\min_k(nrWORKING_k) / \max_k(nrWORKING_k) < fairLevel$: Job nach ausbalanciertem Verfahren ausliefern und Algorithmus beenden.

(6) Falls $\min_k(relDONE_k) < doneRateLowBoost$: Job nach dem Verfahren ausliefern, das neue Benutzer bevorzugt und Algorithmus beenden.

(7) Mit der Wahrscheinlichkeit *powerIndexProb* einen Job nach dem leistungsbasierten Verteilverfahren ausliefern und Algorithmus beenden.

(8) Falls *useUptimes* = true einen Job nach dem betriebszeitbasierten Verfahren ausliefern (mit *s* = *runlengthScale*), ansonsten einen Job nach laufzeitbasiertem Verfahren (ebenfalls mit *s* = *runlengthScale*) ausliefern und Algorithmus beenden.

Durch das Setzen des Parameters *fairLevel* = 1 erzeugt man die rein ausbalancierte Verteilung. Setzt man *fairLevel* = 0 und *doneRateLowBoost* = 1 werden die Jobs ausschließlich mit Bevorzugung neuer Nutzer verteilt. Eine rein leistungsbasierte Verteilung erreicht man mit *fairLevel* = 0, *doneRateLowBoost* = 0 und *powerIndexProb* = 1. Eine rein laufzeitbasierte Verteilung wird mit dem Setzen aller Parameter auf 0 und *useUptimes* = false erzielt, analog dazu eine rein betriebszeitbasierte Verteilung mit *useUptimes* = true.

Die beiden Diagramme in Abb. 3.15 zeigen die beiden Simulationen für die folgende Parametereinstellung:

$$fairLevel = 0{,}1 \quad powerIndexProb = 0{,}05 \quad useUptimes = \text{false}$$
$$doneRateLowBoost = 0{,}03 \quad runlengthScale = \text{dynamisch}$$

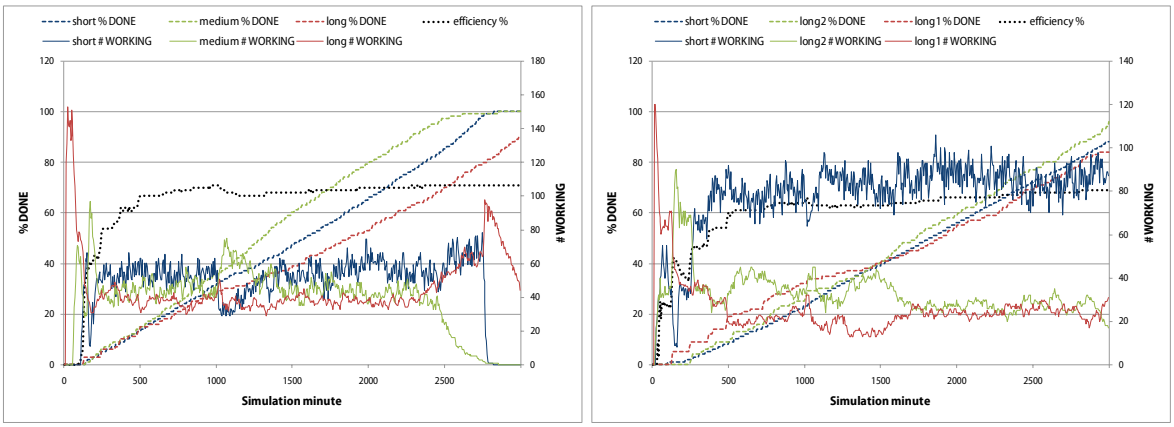


Abb. 3.15 Simulationen A / B bei kombinierter Verteilung

Tabelle 3.11 Resultate bei kombinierter Verteilung

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	64%	49%	Balance entsprechend dem Laufzeitverhalten, gut ausbalanciert / stabil, vorhergehendes Gleichgewicht stellt sich sehr schnell wieder ein
B	61%	42%	Balance entsprechend dem Laufzeitverhalten / stabil, vorhergehendes Gleichgewicht stellt sich schnell wieder ein

Die 5%ige Einstreuung des leistungsbasierten Verfahrens hat den Effekt, dass das System die hohen Effizienz- und Fertigungsraten des laufzeitbasierten Verfahren halten kann (Tabelle 3.11), jedoch wesentlich stabiler und adaptiver auf Zuverlässigkeitsänderungen reagiert. Dass bei der Simulation B wesentlich mehr Rechenknoten für die Kurzjobs arbeiten kommt daher, dass die Knoten in der Mehrheit eben nicht zuverlässig genug sind für die lange laufenden. Entscheidend ist jedoch, dass sich die Knoten, die für die lange

laufenden Jobs zuständig sind, gleichmäßig auf die beiden Langläufer verteilen und dass dieses Verhältnis auch nach dem Umschalten der Zuverlässigkeit erhalten bleibt.

Die kurze Phase zu Beginn, die jeden Jobtyp stark bevorzugt bis 3% Fertigungsrate erreicht sind, unterstützt die beiden leistungs-/laufzeitbasierten Verfahren, da diese nur richtig arbeiten können, wenn man für jeden Knoten und jeden Job-Typ dessen Anforderungen kennt. Dies erreicht man, indem man Jobs, von denen noch nicht viele fertig bearbeitet sind, zunächst einmal bevorzugt behandelt. Das ausbalancierende Verfahren wird hier nur noch mit einem sehr niedrigen Anteil angewendet, bzw. nur noch dann, wenn die beiden effizienzoptimierenden Verfahren aufgrund einer zu homogenen Knotenleistung oder zu ähnlichen Job-Anforderungen nicht sinnvoll einsetzbar wären.

Eine wünschenswerte Parametrisierung wäre jedoch eine, die auf die Einstreuung des leistungsbasierten Verfahrens verzichten könnte und auch ohne die doch recht willkürlich gewählte Spreizung funktionieren würde. Hier zeigt sich die Stärke der betriebszeitbasierten Verteilstrategie. Abb. 3.16 und Tabelle 3.12 zeigen die beiden Simulationsergebnisse bei folgender Parametrisierung:

$$fairLevel = 0{,}1 \quad powerIndexProb = 0 \quad useUptimes = \text{true}$$
$$doneRateLowBoost = 0{,}03 \quad runlengthScale = 0$$

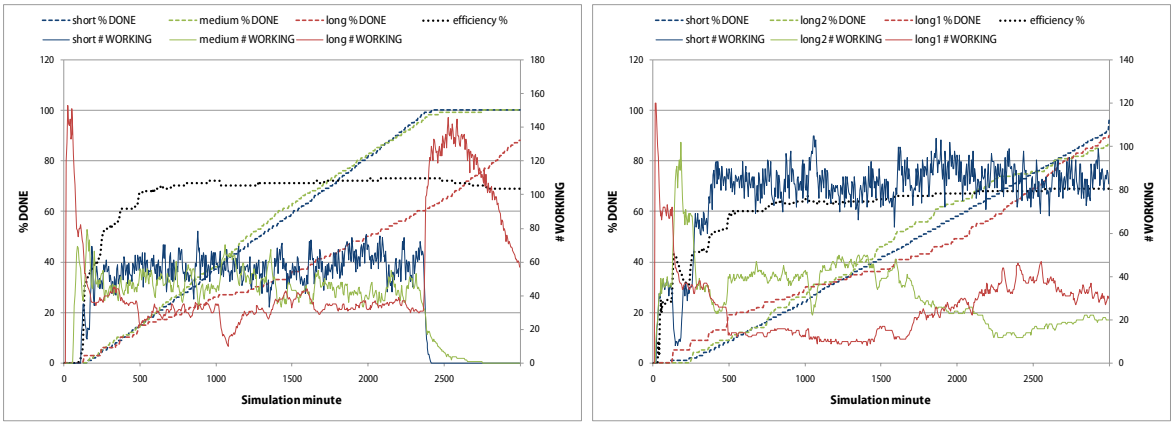


Abb. 3.16 Simulationen A / B bei alternativer kombinierter Verteilung

Tabelle 3.12 Resultate bei alternativer kombinierter Verteilung

Simulation	*avEff*	*avDONE*	Balance / Verhalten bei Zuverlässigkeitsänderung
A	66%	51%	Balance entsprechend dem Laufzeitverhalten, gut ausbalanciert / stabil, vorhergehendes Gleichgewicht stellt sich sofort wieder ein
B	61%	42%	Balance entsprechend dem Laufzeitverhalten / stabil, vorhergehendes Gleichgewicht stellt sich sofort wieder ein

Auf die kurze Bevorzugung neuer Nutzer mit wenigen Prozent kann jedoch auch hier nicht verzichtet werden, da das betriebszeitbasierte Verfahren ebenfalls davon anhängig ist, dass man möglichst schnell die Anforderungen der einzelnen Nutzer kennt. Trotzdem kann auf zwei der administrativ festzulegenden Parameter verzichtet werden, und auf-

grund der Tatsache, dass das Verfahren sehr gute Ergebnisse liefert, wurde für den Praxiseinsatz zunächst genau diese Konfiguration benutzt.

3.3.7 Verbesserung von Simulation und Betriebszeitverfahren

Die bisher durchgeführten Simulationen eignen sich gut, um die prinzipiellen Stärken und Schwächen der einzelnen Verfahren zu analysieren und diese zu einem Gesamtverfahren zusammenzuschalten. In der Praxis ist es nun aber nicht so, dass ein Rechner zu jedem Zeitpunkt mit einer konstanten Wahrscheinlichkeit ausfällt, sondern dass er eine gewisse Zeit zuverlässig arbeitet und dann mit steigender Wahrscheinlichkeit ausfällt. Ein typischer Arbeitsplatzrechner arbeitet z. B. von Beginn der Arbeitszeit an konstant 7–10 Stunden durch, wird danach aber meist heruntergefahren. Dies läuft dann jeden Tag wiederholt gleich oder zumindest ähnlich ab.

Um diesen Umstand simulieren zu können, wurde der im vorigen Abschnitt beschriebene Simulator erweitert, neben den bisher bekannten Ausfallwahrscheinlichkeiten kann nun für jeden simulierten Knoten spezifiziert werden, wie lange zuverlässig gearbeitet werden soll und in welcher Zeitspanne dann die Ausfallwahrscheinlichkeit linear ansteigt. Ein Knoten wird nun durch folgenden XML-Code definiert (Listing 3.6):

Listing 3.6 Verbesserte Knotensimulation

```
<client cnt="1" power="4000" fail="3" fail2="5" zerofp="60" incfp="40"/>
```

Der so simulierte Rechner würde 60 Minuten ohne Ausfallwahrscheinlichkeit arbeiten, in den folgenden 40 Minuten würde diese jedoch auf 3% ansteigen und nach den üblichen 1000 Simulationsminuten auf 5% wechseln. Die restlichen Attribute geben unverändert die Anzahl der Knoten und deren CPU-Leistung an. Abb. 3.17 verdeutlicht das Arbeitsschema dieses Knotens, indem die Ausfallwahrscheinlichkeit in Abhängigkeit der Simulationszeit angegeben ist. Fällt der Knoten aus, beginnt der Zyklus wieder von vorne.

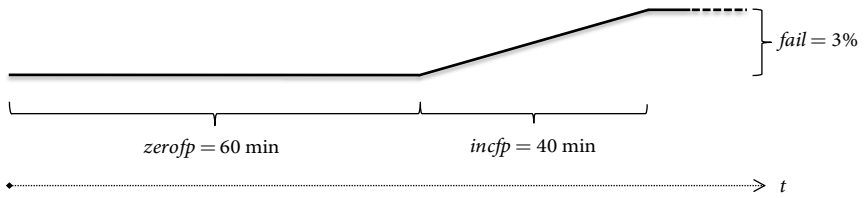


Abb. 3.17 Arbeitsschema eines simulierten Knotens

Das betriebszeitbasierte Verfahren, das sich bei den vorangegangenen Simulationen als das Geeignetste herausgestellt hat, wurde ebenfalls verändert, um dieses realitätsnähere Verhalten der Knoten bei der Verteilung der Jobs besser berücksichtigen zu können. Zunächst wird nicht nur die durchschnittliche Betriebszeit (avU_i) eines Knotens erfasst, sondern auch die jeweilige aktuelle Betriebszeit (acU_i). Man weiß also von einem Knoten nicht nur, wie lange er durchschnittlich verfügbar ist, sondern auch wie lange er davon

schon in Betrieb war. Der Wert $avTARGET$ berechnet sich nun aus diesen beiden Werten, der Zuverlässigkeit R_i und der CPU-Leistung rB_i des Knotens *i* wie folgt:

$$avTARGET' = \begin{cases} avU_i - acU_i, & acU_i \leq avU_i \\ (R_i + 1)(acU_i - avU_i), & acU_i > avU_i \end{cases} \quad \text{mit } R_i \in [-1,1]$$

$$avTARGET = avTARGET' \cdot \sum_{j=1}^{l} rB_j \Big/ l \cdot rB_i \quad \text{mit } l = \#\text{Knoten}$$

Das Verfahren geht also davon aus, dass sich die Ausfallwahrscheinlichkeit eines Rechners stark erhöht, je näher seine aktuelle Betriebszeit an seine bisher ermittelte durchschnittliche Betriebszeit heranrückt. Je enger die beiden Werte beieinander liegen, desto vorsichtiger werden die Jobs verteilt: Ein Rechner, der im Schnitt 8 Stunden arbeitet, davon aber schon 7½ gelaufen ist, sollte besser nur noch einen 30-Minuten Job erhalten. Werden diese kurzen Jobs dann aber zuverlässig ausgeführt, so erhält der Knoten nach und nach wieder längere Aufträge, skaliert mit seiner relativen CPU-Leistung.

Aus dem *avTARGET* -Wert wird dann wie bisher auch der *RLTV* -Wert bestimmt, indem von den durchschnittlichen Joblaufzeiten die jeweiligen Mittelwerte $avTM_i$ berechnet, und anschließend die *avTARGET* am nächsten liegenden Werte $avT*$ und $avTM*$ ermittelt werden. An den anfragenden Knoten ausgeliefert wird auch hier wiederum der Job J_i, der $\left| RLTV - avT_{jT(i)} \right|$ minimiert. So passt das System die Länge der an einen Knoten verteilten Jobs besser an dessen Betriebszeitmuster an, berücksichtigt dabei seine CPU-Leistung im Vergleich zu den anderen Knoten, und reagiert gleichzeitig auch angemessen auf Rechner, die ihre Arbeitszeit oder Ausfallwahrscheinlichkeit ändern.

Um dieses neue Verfahren gegenüber den bisher entwickelten Strategien zu testen, wurde mit dem verbesserten Simulator die folgende Konfiguration A simuliert (Listing 3.7).

Listing 3.7 Verbesserte Simulationskonfiguration A

```
<simConfig> <!--Simulation A-->
 <clients>
  <client cnt="10" power="4000" fail="0" fail2="0" zerofp="3000" incfp="300"/>
  <client cnt="30" power="9000" fail="3" fail2="3" zerofp="60" incfp="120"/>
  <client cnt="30" power="9000" fail="2" fail2="2" zerofp="120" incfp="180"/>
  <client cnt="10" power="9000" fail="5" fail2="5" zerofp="400" incfp="60"/>
  <client cnt="10" power="14000" fail="5" fail2="5" zerofp="400" incfp="60"/>
 </clients>
 <simulation>
  <step cnt="4000" jobtype="short" jobduration="10" steps="0"/>
  <step cnt="300" jobtype="medium" jobduration="90" steps="0"/>
  <step cnt="60" jobtype="long" jobduration="360" steps="2990"/>
 </simulation>
</simConfig>
```

Es wurden also neben absolut zuverlässigen Knoten auch solche simuliert, die nur vergleichsweise kurz zuverlässig arbeiten und ihre Ausfallwahrscheinlichkeit dann langsam steigern, sowie Knoten, die fast einen kompletten Arbeitstag durchlaufen, danach aber recht schnell abgeschaltet werden. Auf ein Umschalten der Ausfallwahrscheinlichkeit

wurde verzichtet. Bei den simulierten Jobs wurden mit Laufzeiten von 10, 90 und 360 Minuten praxisnahe Erfahrungswerte gewählt.

Für die folgenden Messungen wurde auch das Verfahren zur Effizienzbestimmung geändert. In deren Berechnung fließen nun nicht mehr alle über den gesamten Simulationszeitraum erfassten erfolgreichen bzw. erfolglosen Arbeitsminuten, sondern nur die jeweils aktuellsten Werte ein. Nach jeweils $2l$ Arbeitszeitaktualisierungen (l entspricht der Anzahl der Rechenknoten) wird von den gespeicherten Werten die jeweils ältere Hälfte gelöscht. Der Vorteil liegt darin, dass man nun genauer feststellen kann, wie effizient das Verfahren zu einem bestimmten Zeitpunkt arbeitet. Zusätzlich wurde der sogenannte *Makespan* bestimmt: Die absolute Dauer der Simulation, bis alle vorhandenen Jobs bearbeitet sind.

Abb. 3.18 zeigt die Simulation beim ausbalancierten Verfahren (links) und dem leistungsbasierten Verfahren (rechts). Im Gegensatz zu den obigen Messläufen arbeitet das leistungsbasierte Verfahren besser: Die durchschnittliche Effizienz beträgt zwar nur 61% gegenüber 60% beim ausbalancierenden Verfahren, jedoch fällt positiv auf, dass am Ende der Simulation praktisch alle Jobs erledigt sind, während bei der ausbalancierten Verteilung am Ende noch 32% der lang laufenden Jobs zu bearbeiten sind.

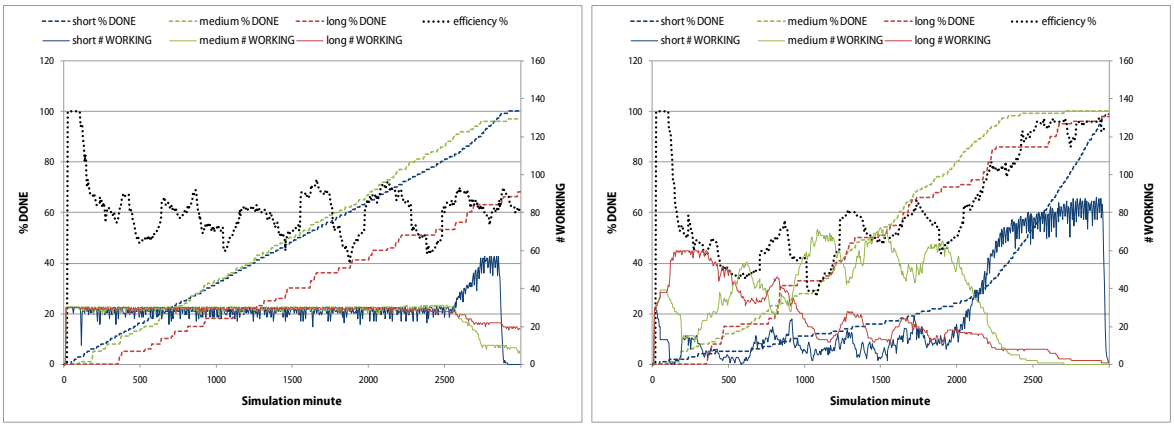


Abb. 3.18 Praxisnahe Simulation ausbalanciert und leistungsbasiert

Der (in den Abbildungen nicht ersichtliche) Makespan von 3765 Simulationsminuten beim ausbalancierten Verfahren sinkt auf einen Wert von 3135. Unter realen Bedingungen würde also eine Zeitspanne von rund 10 Stunden eingespart werden.

Verwendet man stattdessen das kombinierte Verteilverfahren mit den im vorigen Abschnitt als sinnvoll ermittelten Parametrisierungen

$$fairLevel = 0{,}1 \quad powerIndexProb = 0{,}05 \quad useUptimes = \text{false}$$
$$doneRateLowBoost = 0{,}03 \quad runlengthScale = \text{dynamisch}$$

für den Einsatz des laufzeitbasierten bzw. mit

$$fairLevel = 0{,}1 \quad powerIndexProb = 0{,}0 \quad useUptimes = \text{true}$$
$$doneRateLowBoost = 0{,}03 \quad runlengthScale = (\text{wird ignoriert})$$

für die Verwendung des verbesserten betriebszeitbasierten Verfahrens, so ergibt sich eine weitere Effizienzsteigerung gegenüber dem ausbalancierten Verfahren (Abb. 3.19 links). Der Makespan beträgt hier 3460 Simulationsminuten.

Das im rechten Teil dieser Abbildung gezeigte neue Betriebszeitverfahren steigert die durchschnittliche Effizienz auf 65%. Auch hier sind die Jobs deutlich früher fertig bearbeitet als beim ausbalancierten Verfahren, der Makespan beträgt 3285. Alle drei Verteilverfahren verhalten sich bei der Simulation ähnlich: Die Mehrzahl der Rechenknoten arbeitet für die 90-Minuten Jobs, während die sehr kurzen und die sehr langen Jobs auf die schlechtesten bzw. die besten Knoten verteilt werden. Beim neuen Betriebszeitverfahren werden die kurz laufenden Jobs dazu benutzt, die Restlaufzeiten der Knoten mit Arbeit auszufüllen.

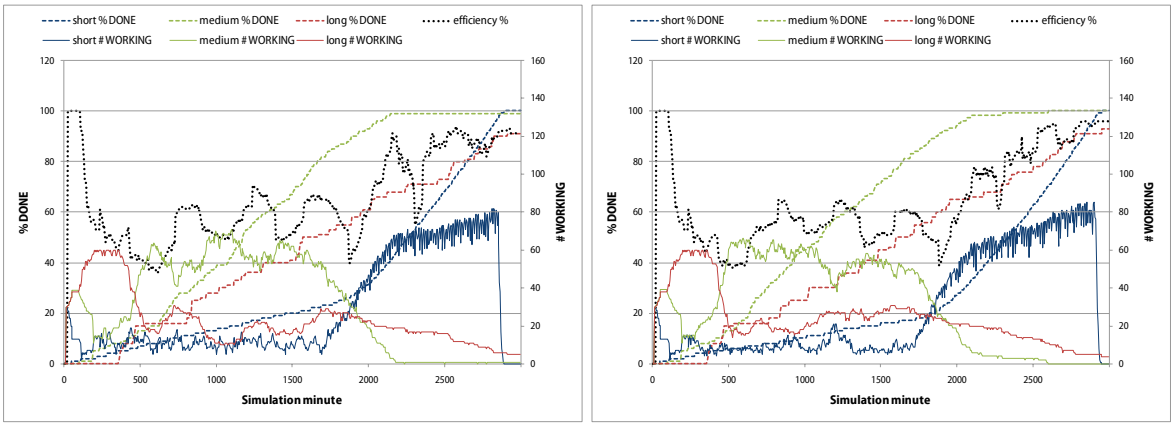


Abb. 3.19 Praxisnahe Simulation laufzeit- und betriebszeitbasiert

Um die positiven Erfahrungen zu bestätigen, wurden zwei weitere Simulationskonfigurationen B und C mit gleicher Knotenkonfiguration getestet. Die beiden folgenden XML-Fragmente (Listings 3.8 und 3.9) zeigen die Jobkonfigurationen dieser Simulationen, auf die Angabe der Knotenkonfiguration wurde hier verzichtet.

Listing 3.8 Verbesserte Simulationskonfiguration B

```
<simulation> <!--Simulation B-->
  <step cnt="2000" jobtype="job_10" jobduration="10" steps="0"/>
  <step cnt="1500" jobtype="job_30" jobduration="30" steps="0"/>
  <step cnt="150" jobtype="job_120" jobduration="120" steps="0"/>
  <step cnt="150" jobtype="job_240" jobduration="240" steps="2990"/>
 </simulation>
```

Listing 3.9 Verbesserte Simulationskonfiguration C

```
<simulation> <!--Simulation C-->
  <step cnt="2500" jobtype="job_20" jobduration="20" steps="0"/>
  <step cnt="150" jobtype="job_180" jobduration="180" steps="0"/>
  <step cnt="150" jobtype="job_220" jobduration="220" steps="2990"/>
</simulation>
```

Vergleicht man die mit den Verteilstrategien jeweils erreichten Werte für den Makespan der drei Simulationen, so ergibt sich ein eindeutiges Bild (Abb. 3.20): Alle drei effizienzoptimierenden Verfahren sind im Schnitt rund 600 Simulationsminuten besser als die reine Ausbalancierung. Aufgrund der im vorigen Abschnitt dargestellten potentiellen Probleme des Leistungs- und des Laufzeitverfahrens wird im Praxisbetrieb das (neue) Betriebszeitverfahren eingesetzt.

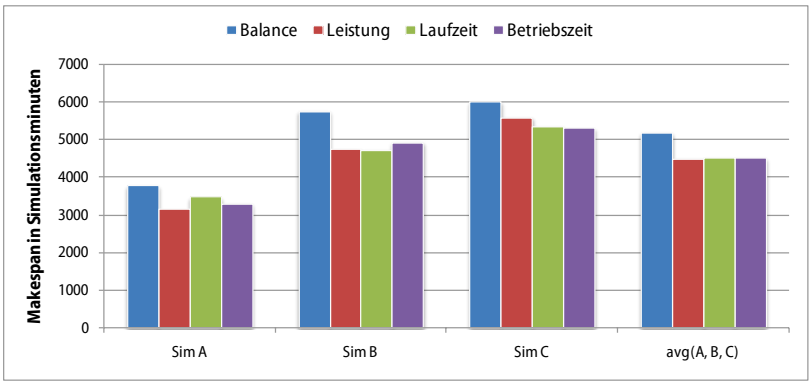


Abb. 3.20 Makespan-Werte der Verteilverfahren

3.3.8 Mehrfachauslieferung von Jobs

Die bisher durchgeführten Simulationen zur Beurteilung der Verteilstrategien haben eine Eigenschaft, die ein Problem bei der Verteilung verdeckt: Bei allen Tests (außer den Makespan-Messungen) war die Gesamtanzahl der zur Berechnung verfügbaren Jobs zu jedem Simulationszeitpunkt immer signifikant höher als die Anzahl der simulierten Rechenknoten. Will man nur die Effizienz oder das Fairnessverhalten einer Verteilstrategie beurteilen, ist das unkritisch, aber was passiert, wenn man am Ende immer weniger verfügbare Jobs auf eine konstant hohe Anzahl sehr heterogener Rechner verteilen muss?

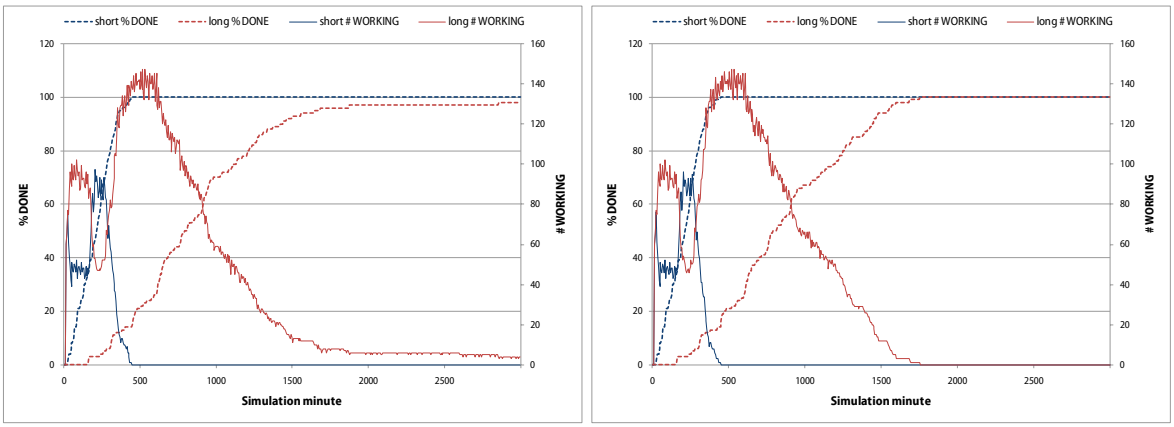


Abb. 3.21 Auswirkung der Mehrfachverteilung von Jobs

Das linke Diagramm in Abb. 3.21 zeigt das Problem. Simuliert wurden zwei Jobtypen zu jeweils 20 und 150 Simulationsminuten Laufzeit. Solange noch Jobs beider Typen vorhanden sind, arbeitet der Scheduler wie gewohnt: Die Jobs werden entsprechend der am

Scheduler eingestellten Verteilstrategie (vgl. Abschnitte 3.3.5ff) an die Rechner verteilt. Sind dann aber irgendwann nur noch lange laufende Jobs verfügbar, werden diese natürlich auch an die schwachen Rechner verteilt, was solange gut funktioniert, wie die starken Rechner ebenfalls arbeiten können. Wenn aber irgendwann die Anzahl der freien Jobs so weit gesunken ist, dass nicht mehr alle Knoten beschäftigt werden können, passiert es, dass die zuverlässigen Rechner nichts mehr tun, weil die Jobs auf den unzuverlässigen Rechnern laufen, die sie dann aber eventuell nicht komplett ausführen können. Es tritt ein unschöner Effekt ein: Die Jobs eines Benutzers werden nicht fertig, obwohl genügend potente Knoten vorhanden sind.

Dieses Problem tritt unabhängig von der gewählten Verteilstrategie auf, da alle Strategien einem Knoten immer einen Job zuteilen, sofern einer verfügbar ist. Eine naheliegende Lösung des Problems wäre, in dieser Situation an schwache Rechner keine Jobs mehr zu verteilen. Dann stellt sich jedoch die Frage nach einer sinnvollen Parametrisierung:

- Bei welchem Verhältnis von freien Jobs zur Anzahl an Rechenknoten verweigert man einem schwachen Rechner einen Job?
- Wie wird „schwacher Rechner“ definiert?
- Was passiert, wenn man einem unzuverlässigen Knoten die Auslieferung eines Jobs verweigert und dieser Rechner jedoch der einzige ist, der (z. B. aufgrund von Betriebssystemanforderungen des Jobs) diesen berechnen könnte?
- Verweigert man einem schwachen Knoten immer einen Job oder nur mit einer bestimmten Wahrscheinlichkeit? Und wie wird diese Wahrscheinlichkeit quantifiziert?

Um diesen nicht sinnvoll und universell lösbaren Problemen aus dem Weg zu gehen, wird ein einfacherer Mechanismus genutzt, der sicherstellt, dass

(a) die zuverlässigen Rechner zu jedem Zeitpunkt Jobs bearbeiten und

(b) keinem Rechner die Jobausführung verweigert wird, so dass Jobs mit potentiell exotischen Anforderungen nicht unnötig zurückgehalten werden.

JoSchKa erreicht das dadurch, indem die gerade laufenden Jobs mehrfach an Agenten ausgeliefert werden. Sobald kein freier Job mehr verfügbar ist, wird die Kandidatenliste für den Scheduler aus denjenigen Jobs aufgebaut, die den Status WORKING besitzen und bei denen der Zwischenergebnisupload nicht aktiviert ist (vgl. 3.3.2). Jobs mit aktiviertem Zwischenergebnisupload sollten nicht mehrfach gleichzeitig ausgeführt werden, da möglicherweise unterschiedliche, aus verschiedenen zeitlich nicht zusammenpassenden Berechnungszuständen des Jobs stammende Zwischenergebnisse zum Server übertragen werden würden. Aus dieser Kandidatenliste bestimmt dann der Scheduler nach den in den vorangegangenen Abschnitten beschriebenen Verfahren wie gewohnt denjenigen Job, der an den anfragenden Agenten ausgeliefert wird.

Das rechte Diagramm der Abb. 3.21 zeigt deutlich den Vorteil der Mehrfachauslieferung: Nach ca. 1800 Simulationsschritten sind alle Jobs erledigt, wohingegen ohne Mehrfachauslieferung zu diesem Zeitpunkt erst 96% der lang laufenden Jobs erledigt sind und – das

ist das entscheidende – sich diese Rate bis zum Ende der Simulation auch nicht wesentlich verbessert: Nach 3000 Simulationsschritten sind erst 98% erreicht.

Effizienzbetrachtungen wurden bei den Tests zur Mehrfachauslieferung nicht durchgeführt, da es leicht passieren kann, dass ein Rechner unnötig arbeitet, obwohl er einen Job korrekt zu Ende gebracht hat. Durch die redundante parallele Berechnung wurde der gleiche Job möglicherweise schon vorher korrekt abgeschlossen. Ebenso kann es passieren, dass ein Rechner einen redundant ausgeführten Job nicht zu Ende bringt, das aber nicht weiter schlimm ist, da der Job ja gleichzeitig noch von einem anderen Knoten (möglicherweise erfolgreich) bearbeitet wird. In dieser Situation die Effizienz sinnvoll zu quantifizieren, gestaltet sich schwierig, zumal die Scheduling- und Monitoring-Komponenten des Systems völlig unabhängig voneinander arbeiten. In die vom Knoten erfassten Leistungsmerkmale (vgl. Abschnitt 3.3.4) fließen die durch Mehrfachausführung verwendete Arbeitszeit und die eventuellen Ausfälle jedoch ein, was bedeutet, dass die Werte avF, avS, avU, R und nP nach wie vor ständig gemessen und aktualisiert werden.

3.4 Agent

Beim JoSchKa-Agenten handelt es sich um den Teil des Systems, der für die Ausführung der Jobs auf den teilnehmenden Rechenknoten verantwortlich ist. Das Programm ist so konzipiert, dass es ohne Nutzerinteraktion auskommt und deshalb auf dem ausführenden Rechner als Hintergrunddienst gestartet werden kann. Selbsttätig erledigt er die folgenden Hauptaufgaben:

(1) Feststellen, welche Art von Jobs der Rechner, auf dem der Agent läuft, ausführen kann. Das betrifft im Wesentlichen die Frage, ob die Java-Laufzeitumgebung installiert ist und ob verschiedene Skriptsprachen, wie Python oder Perl, zur Verfügung stehen. Da der Agent betriebssystemunabhängig ist, wird auch das Betriebssystem festgestellt.

(2) Beim Server über die SOAP-Schnittstelle einen Job erfragen. Sofern einer vorhanden ist, werden danach die zugehörigen Dateien über die Download-Schnittstelle heruntergeladen.

(3) Den Job starten und auf das Ende warten, dabei regelmäßig über einen Hintergrund-Thread den Server (über SOAP) kontaktieren, den Status mitteilen und den Job gegebenenfalls abbrechen, falls dies vom Server gefordert wird. Dabei werden die Konsolen- und Fehlerausgaben des Jobs abgefangen, sofern vom Benutzer gewünscht.

(4) Nach Beendigung des Rechenjobs alle vorhandenen Ergebnisdateien über die Upload-Schnittstelle zum Server hochladen, sofern sich der Job selbst fehlerfrei beendet hat. Gegebenenfalls sind Konsolen- und Fehlerausgaben in separate Dateien zu schreiben und dann ebenfalls zum Server zu übertragen.

(5) Sofern alle Dateien beim Server sind und der Job nicht abgebrochen wurde, eine abschließende Bestätigung zum Server schicken.

Diese Punkte werden im Folgenden näher erläutert.

3.4.1 Systemumgebung des Agenten

Ein wesentlicher Aspekt von JoSchKa ist die Unabhängigkeit von Programmiersprachen, Laufzeitumgebungen, Betriebssystemen oder Hardware. Die einzelnen Jobs dürfen also unterschiedlichste Anforderungen bezüglich der Software des Zielsystems besitzen. Damit ein Job auch nur auf einem System gestartet wird, auf dem er (aus Software-Sicht) lauffähig ist, stellt der Agent einmalig fest, auf welchem Betriebssystem er läuft, welche Laufzeitumgebungen vorhanden sind und welche systemfremden Skriptsprachen vorhandenen sind. Da es gerade davon sehr viele gibt, werden nur die gängigsten Skriptsprachen getestet. Für die Tests der eigenen Plattform wurden Zeichen festgelegt, aus welchen der Agent im Erfolgsfalle die Softwareplattformbeschreibung erzeugt. Der Agent wurde in C# realisiert, benötigt zur Ausführung also zwingend eine entsprechende Ablaufumgebung. Über die Standardbibliothek dieser Laufzeitumgebung wird das Betriebssystem festgestellt (in Klammern steht jeweils das für die Plattformbeschreibung verwendete Zeichen):

- Windows (W)
- Linux (L)

Im Falle von Windows wird davon ausgegangen, dass die Laufzeitumgebung das Microsoft .NET-Framework (N) ist, sollte das Betriebssystem Linux sein, wird die von Novell unterstützte Mono-Laufzeitumgebung (M) angenommen. Danach werden die folgenden Tests durchgeführt:

- Laufzeitumgebung Java (J)
- Skriptsprachen Perl (P) und Python (Y)
- Statistiksystem R (R)
- General Algebraic Modeling System GAMS (G)

Diese Tests werden durchgeführt, indem der Agent vom Betriebssystem die Befehle *java*, *perl*, *python*, *R* und *gams* ausführen lässt und die gegebenenfalls vom System erzeugte Fehlermeldung abfängt und auswertet. Sollte der Agent beispielsweise unter Windows ausgeführt werden und ist dort ein Python-Interpreter installiert, der mit dem Kommando *python* angesprochen werden kann, so lautet die Softwareplattformbeschreibung für diesen Agenten „WNY“. Ein Java- und Perl-fähiges Linux würde mit „LMJP“ beschrieben werden, wobei die Reihenfolge der Zeichen keine Rolle spielt.

In einem weiteren Schritt stellt der Agent fest, wie viel physikalischer Hauptspeicher (in Megabyte) auf dem Rechensystem installiert ist und hängt diesen Wert mit einem Punkt abgetrennt an die Softwareplattformbeschreibung an. Mit einem weiteren Punkt getrennt wird die maximale Datenmenge (ebenfalls in Megabytes) angegeben, die der Agent bei Beendigung eines Jobs an den Server übertragen kann. Da dieser Wert von der Netzwerkbandbreite und von administrativen Vorgaben abhängen kann, wird hier zunächst eine Wildcard (‚*‘) eingetragen, was aber vom Betreiber eines Agenten auch weiter eingeschränkt werden kann (vgl. Abschnitt 4.5). Der letzte, wiederum von einem Punkt abgetrennte Wert beschreibt die vorhandene nutzbare Prozessorarchitektur, ermittelt durch Auslesen der Umgebungsvariablen *CPU_ARCHITECTURE*.

Die vollständige Plattformbeschreibungssyntax lautet also:

```
<Software>.<Hauptspeicher>.<Uploaddatenmenge>.<CPU-Architektur>
```

Ein Java-fähiges Windows mit 1024 MB Hauptspeicher auf einem 32-Bit Rechner würde also in der kompletten Plattformbeschreibung „WNJ.1024.*.x86“ resultieren.

Zusätzlich zur Plattform wird beim Start des Agenten einmalig die Leistung des Systems gemessen und zum Server geschickt. Diese Daten werden beim Scheduling verwendet (siehe Abschnitte 3.3.4ff). Weiterhin wird die Netzwerkbandbreite durch den Download zweier fest definierter Dateien vom Server gemessen. Sollte der Download zu lange dauern, wird das dem Server bei der Jobanfrage mitgeteilt, so dass nur Jobs ausgeliefert werden, bei denen wenige Daten heruntergeladen werden müssen.

3.4.2 Anfragen und Downloaden eines Jobs

Über die *GetJob()* Webmethode des Servers teilt der Agent dem Server seinen Arbeitswunsch mit. Mit dieser Schnittstelle spezifiziert der Agent, welche Art von Jobs er ausführen kann. Neben der im vorigen Abschnitt erläuterten Plattformbeschreibung ist das unter anderem eine Zeichenkette, mit der der Agent präferierte Benutzer bzw. Jobtypen angeben darf (die Zeichenkette kann auch leer bleiben, für beliebige Typen). Weiterhin muss der Agent eine maximale Datenmenge angeben, die er downloaden will („−1“ für beliebige Datenmengen), und den Namen des Rechners, auf dem er läuft. Dieser Name enthält auch den Namen des Benutzerkontos unter dem der Agent betrieben wird, einen Hashwert über verschiedene Seriennummern der Rechnerhardware sowie die ID des Agentenprozesses, so dass der Server mehrere auf dem gleichen Rechner laufende Agenten unterscheiden und Rechner mit wechselnden IP-Adressen wiedererkennen kann. Eine beispielhafte Anfrage lautet wie folgt (vgl. auch 3.3.2):

```
GetJob("", "WNJ.1024.*.x86", "aifbacheron[85fd8aa7]\bonn.1286", "-1")
```

Der Server liefert als Antwort ein XML-codiertes Zeichenketten-Array, das die Jobbeschreibung enthält. Im Einzelnen sind das zunächst eine Statusanzeige, die Werte der serverinternen Datenfelder *jobType*, *jobID*, *command*, *resultfiles*, *files*, *tempupload*, *rndID* und *userIdentifier* sowie die MD5-Hashwerte der in *files* angegebenen Dateien. Eine beispielhafte Antwort des Servers würde wie folgt aussehen (Listing 3.10):

Listing 3.10 SOAP Antwortdaten auf Jobanfrage

```
<ArrayOfString>
     <string>JOB_OK</string>
     <string>mbo_bench</string>
     <string>03211620-90803507180800001</string>
     <string>bench.exe a 11 33</string>
     <string>results\output.log</string>
     <string>bench.exe</string>
     <string>NO</string>
     <string>9247</string>
     <string>run1</string>
     <string>1729D789F0396B2A01FF553F36EA941C</string>
</ArrayOfString>
```

Anschließend prüft der Agent, ob eventuell vorhandene lokale Dateien namensgleich mit den in *files* angegebenen Namen sind. Sollte das der Fall sein und die MD5-Summe der lokalen Datei mit der vom Server mitgeschickten Prüfsumme übereinstimmen, kann aus Effizienzgründen auf den Download verzichtet (vgl. Abschnitt 4.3.3) werden. Andernfalls werden alle Quelldateien per HTTP heruntergeladen, die URL bildet sich aus dem Jobtyp und dem Dateinamen, im Beispiel wäre das:

```
http://<ServerURL>/FileDownload/mbo/bench/bench.exe
```

Der Unterstrich im Jobtyp des obigen Beispiels („mbo_bench“) wird also durch den URL-Pfadtrenner („…mbo/bench…“) ersetzt. Die Dateien werden im lokalen Cacheverzeichnis abgelegt und danach ins Arbeitsverzeichnis kopiert. Der Agent kann auf dem Rechensystem mit eingeschränkten Benutzerrechten betrieben werden, er benötigt lediglich ein einziges Verzeichnis mit Vollzugriff, in dem die Dateien des Rechenjobs abgelegt werden. Der eigentliche Rechenjob wird dann ebenfalls mit eingeschränkten Benutzerrechten gestartet, so dass potentielle Sicherheitsprobleme von vornherein vermieden werden können (vgl. Abschnitt 3.5.1).

3.4.3 Ausführen eines Jobs

Sind alle spezifizierten Dateien heruntergeladen und ins Arbeitsverzeichnis kopiert, startet der Agent die vorgegebene Befehlszeile, im Beispiel wäre das

```
bench.exe a 11 33
```

Dabei wird die Prozesspriorität auf die vom Betriebssystem unterstützte Minimal-Priorität gesetzt[9]. Unter Windows übernimmt das die .NET-Laufzeitbibliothek, mit der der Befehl gestartet wird, unter Linux erreicht man eine geringe Priorität indem der Agent den Befehl mit dem *nice*-Präfix startet:

```
nice -n 19 bench.exe a 11 33
```

Während der Ausführung des Jobs überwacht der Agent ständig den Hauptspeicherverbrauch und die (CPU-)Systemlast und teilt dem Server diese Werte periodisch per SOAP mit. Die zuständige Webmethode heißt *ExchangeStatus()* und hat zusätzlich den Zweck, den *countdown* des Jobs wieder auf den Maximalwert zu setzen (siehe Abschnitt 3.3.3). Die Antwort des Server lautet in jedem Fall *OK* oder *KILL_JOB*, letztere würde den Agenten dazu veranlassen, den Job abzubrechen, wozu die Systembefehle *taskkill* (unter Windows) bzw. *kill* (unter Linux) benutzt werden. Das geschieht z. B. dann, wenn der Nutzer auf dem Server einen Job löscht oder blockiert. In diesem Falle ist es nicht sinnvoll, den Job weiterhin auszuführen. Sollte die gemessene Systemlast zu hoch sein oder der freie Hauptspeicher zur Neige gehen, bricht der Agent den Job selbsttätig ab.

Falls der Benutzer einen temporären Zwischenupload der Ergebnisse wünscht (*tempupload = YES*) werden zufallsgesteuert alle 20–40 Minuten die eventuell vorhandenen Ergebnisdateien zum Server übertragen (siehe Abschnitt 3.4.4).

[9] Der Scheduler des Betriebssystems stellt sicher, dass der Rechenprozess nur dann CPU-Zeit erhält, wenn auf dem System sonst keinerlei Aktivitäten stattfinden. Von der permanenten Hintergrundlast bemerkt der Benutzer normalerweise nichts (siehe auch Abschnitt 4.6).

3.4.4 Upload der Ergebnisse

Beendet sich der vom Agenten gestartete Prozess selbst mit dem Exit-Code 0, wird das als erfolgreich abgeschlossener Lauf gewertet. Die Konsolen- und Fehlerausgaben des Jobs werden in eine Logdatei geschrieben und mit den anderen vom Job neu erzeugten Ergebnis- bzw. geänderten Eingabedateien über die HTTP-Upload-Schnittstelle zum Server übertragen. Jede Datei wird im POST-Teil der Anfrage zip-komprimiert binär angehängt, während im Query-String der URL einige Verwaltungsdaten des Jobs angegeben werden: Die *jobID*, der *jobType*, ein Parameter, der anzeigt ob es sich um den finalen oder nur um einen temporären Upload handelt und eine Pfadangabe:

```
http://<ServerURL>/FileUpload.aspx?id=03211620-
90803507180800011234&type=mbo_bench&isFinalUpload=YES
&targetDir=results
```

Sollte der Agent das Ergebnis aus einem lokalen Unterverzeichnis zum Server schicken, sorgt diese Pfadangabe dafür, dass der Server das Ergebnis ins gleiche Unterverzeichnis ablegt bzw. dieses Verzeichnis anlegt, sollte es noch nicht vorhanden sein. Damit bildet sich auf dem Server die gleiche Dateisystemstruktur der Ergebnisdateien, die bei den Rechenknoten während der Jobausführung ebenfalls so vorhanden war.

3.4.5 Jobausführung abschließen

Nachdem alle finalen Dateiuploads erfolgreich abgeschlossen sind, erfolgt ein letzter SOAP-Aufruf (*CommitJob()*), der dem Server mitteilt, dass der Job nun vollständig erfolgreich erledigt wurde. Bei diesem Aufruf übergibt der Agent dem Server unter anderem die *jobID*, den *jobType* und die Sicherungs-ID *rndID* des Jobs. Nur wenn diese Werte mit den serverintern gespeicherten Werten übereinstimmen, markiert der Server den *status* des Jobs als *DONE*. Damit ist die Bearbeitung dieses Jobs abgeschlossen.

3.5 Sicherheitsaspekte

Stellt man seinen PC dem JoSchKa-System als Rechenknoten zur Verfügung, möchte man natürlich eine gewisse Sicherheit haben, dass die nun potentiell auf dem Rechner ausgeführten Programme und Skripte keinen Schaden anrichten (können). Im Folgenden werden einige Konzepte erläutert, wie sich diese Gefahr vermeiden oder zumindest verringern lassen könnte und warum man als JoSchKa-Nutzer auf vertrauenswürdige Knotenbetreiber angewiesen ist.

3.5.1 Sandbox beim Rechenknoten

Sofern die eigentlichen parallelen Programme (wie das JoSchKa-System selbst) ein .NET-Laufzeitsystem voraussetzen, besteht die Möglichkeit, sie in einer Art Sandbox ablaufen zu lassen. Man kann lokale Richtlinien definieren, was ein per HTTP heruntergeladenes Programm darf und was nicht. Die .NET-Laufzeitumgebung kann z. B. so konfiguriert werden, dass ein solches Programm etwa einen bestimmten Teil der Klassenbibliothek nicht nutzen darf, dass es keine Subprozesse erzeugen kann oder ihm die TCP/IP-Kommunikation verboten ist. Will man jedoch auch nativen Programmcode

(etwa in C/C++ geschrieben) ausführen, stehen solche Mechanismen nicht zur Verfügung und man ist auf die Sicherheitsfeatures des ausführenden Betriebssystems angewiesen. Hauptsächlich sind das Dateisystemberechtigungen, unter Windows kommt noch die Zugriffssteuerung der Registry dazu. Grundsätzlich liegt es also am Administrator des Rechenknotens, festzulegen, was ein heruntergeladenes Programm darf und was nicht. Diese Verantwortung kann ihm nicht abgenommen werden und ein gewisses Vertrauen in die Entwickler der Rechenprogramme ist notwendig. Das JoSchKa-System unterstützt ihn in der Hinsicht, dass der Agent auf dem Rechenknoten mit sehr eingeschränkten Benutzerrechten lauffähig ist und auch mit solchen eingeschränkten Rechten gestartet werden sollte. Ist dies der Fall, werden alle vom Agenten gestarteten Rechenprogramme ebenfalls mit diesen eingeschränkten Rechten laufen.

Eine Möglichkeit, für mehr Sicherheit zu sorgen, wäre die Einführung von digitalen Codesignaturen, bei denen jedes Rechenprogramm (bzw. seine Bestandteile) mit einem vertrauenswürdigen Schlüssel signiert wird und der Agent vor dem Start eines solchen Programms die Gültigkeit der Signatur überprüft. Dies wurde jedoch im Rahmen dieser Arbeit nicht realisiert (vgl. Abschnitt 5.2.3).

Eine zuverlässige Möglichkeit, eine wirksame Sandbox aufzubauen, wäre es, den Agenten innerhalb eines virtuellen Rechners zu starten. Beispiele für Virtualisierungssoftware wären z. B. VMWare oder VirtualPC. Die absolute Sicherheit einer solchen Sandbox erkauft man sich jedoch mit gewissen Leistungseinbußen und der Tatsache, dass für das Gastsystem eventuell Softwarelizenzgebühren anfallen.

3.5.2 Verschlüsselung

Es liegt nahe, aus Sicherheitsgründen den kompletten HTTP-Datentransfer über SSL/TLS durchzuführen, dies hätte jedoch einige Nachteile bei nur geringem Sicherheitsgewinn. Zunächst müsste man dafür sorgen, dass auf jedem Rechenknoten das Serverzertifikat installiert ist und dort auch regelmäßig erneuert wird (Normalerweise laufen die Zertifikate nach einem Jahr ab!). Weiterhin würde eine Verschlüsselung eine erhebliche Belastung des Servers nach sich ziehen, da Verschlüsselungsalgorithmen sehr rechenaufwändig sind. Die Folgen sind offensichtlich: Die Anzahl der Jobs und Agenten, die der Server ohne Performanceverlust gleichzeitig verwalten kann, würde deutlich sinken.

Die Vorteile einer verschlüsselten Übertragung sind indes gering. Man sichert damit lediglich die Datenübertragung, nicht jedoch die Daten, die während der Jobausführung auf dem Rechenknoten liegen. Wenn sich also der Betreiber eines Rechenknotens für die Ergebnisse eines Jobs interessiert, kann man das auch mit einer verschlüsselten Datenübertragung niemals verhindern.

3.6 Schnittstellen und Tools für Entwickler

In diesem Abschnitt werden neben der Vorgehensweise und den Anforderungen, die an eine mit JoSchKa verteilbare Anwendung gestellt werden, auch die Managementprogramme vorgestellt, die den Nutzer dabei unterstützen, wenn es darum geht, seine Anwendung auf die Rechenknoten verteilen zu lassen.

3.6.1 Parallelisierung

Der Entwickler einer parallelen Anwendung hat nur wenige Aufgaben zu erfüllen, damit sich seine Applikation über das vorgestellte System verteilen lässt. Die Wichtigste stellt die Parallelisierung auf Dateisystemebene dar. Das Programm muss sich in parallele Teile aufteilen lassen, von denen jede einzelne Einheit charakterisiert ist durch

- eine Menge von Eingabedateien $E = \{e_1,\ldots,e_n\}$,
- eine Befehlszeile C (oder ein Skript) das die Eingabedateien verarbeitet, und
- eine Menge von Ausgabedateien $A = \{a_1,\ldots a_m\}$, die durch die Befehlszeile oder das Skript erzeugt werden.

Lässt sich ein paralleles Problem in Einheiten der Form $J = (E,C,A)$ zerlegen, kann es mit dem beschriebenen System verteilt werden. Aus diesen Einheiten werden anschließend die einzelnen Rechenjobs erzeugt.

3.6.2 Management-GUI

Spezifikation neuer Jobs

Zunächst muss der Nutzer seiner Anwendung einen Namen geben, z. B. „mopso“. Zusammen mit der Benutzer-ID wird daraus dann später der *jobType* aller Jobs, die zu dieser parallelen Anwendung gehören. Über die Quelldateien-Schnittstelle des Servers findet jeder Nutzer ein Verzeichnis vor, das so lautet wie seine Benutzer-ID, z. B. „smo“.

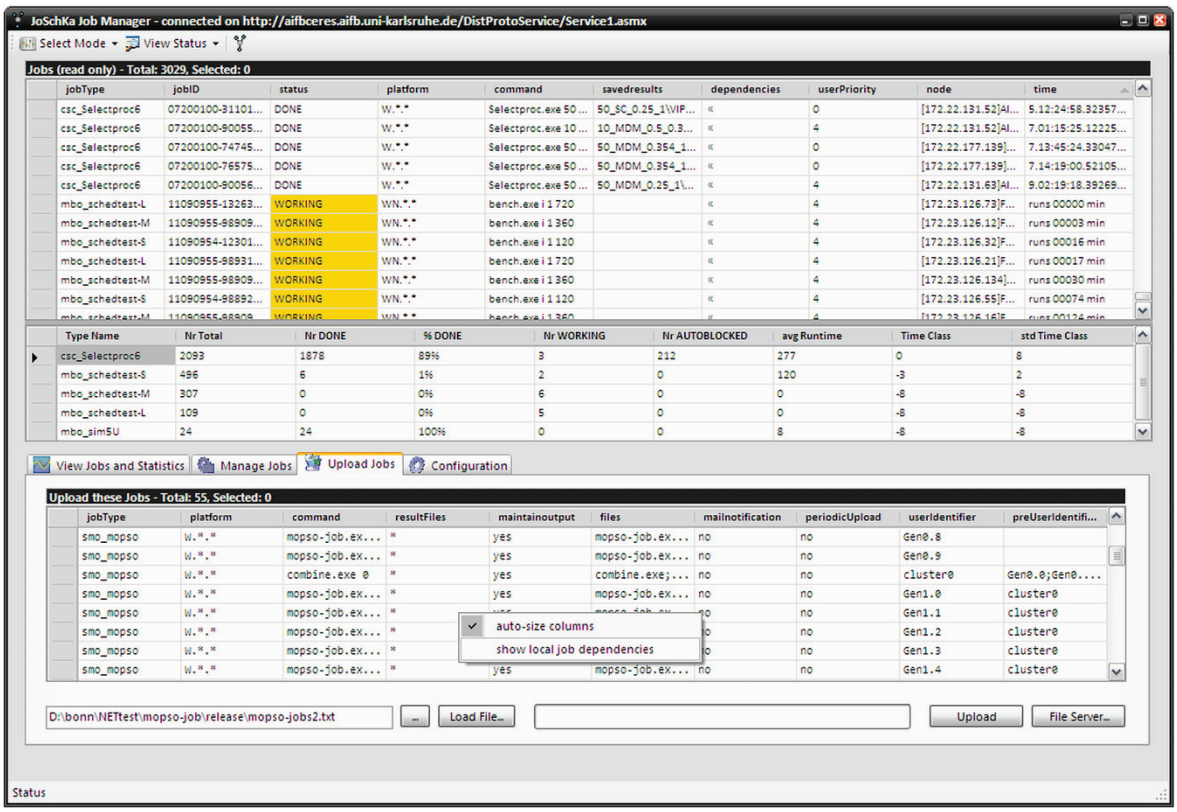


Abb. 3.22 Jobmanagement-Tool: Upload

In diesem Verzeichnis muss ein Unterverzeichnis angelegt werden, das genauso heißt wie die parallele Anwendung selbst, in diesem Fall also „mopso“. In dieses Verzeichnis sind alle Eingabedateien zu kopieren. Dazu gehören auch die Programm- oder Scriptdateien selbst (.exe, .class, .cmd, …), denn sie werden ebenfalls vom Agenten benötigt. Aus diesem Verzeichnis wird der Agent später über den Webserver die Dateien herunterladen.

In einem zweiten Schritt muss der Entwickler seine einzelnen Jobs zusammenstellen und zum Server laden. Dazu wird das Management-Tool *DistProtoManager.exe* benutzt (Abb. 3.22). Es zeigt in den oberen beiden Tabellen den Zustand der Jobs sowie statistische Daten über die einzelnen Job-Typen. In die untere Tabelle des Reiters *Upload Jobs* müssen die einzelnen Jobdaten eingetragen werden. Pro Job ist eine Zeile vorgesehen. Die Spalten bedeuten im Einzelnen:

- *jobType:* Der Typ des Jobs. Er setzt sich aus Benutzer-ID und Anwendungsname zusammen: <Benutzer-ID>_<Anwendungsname>. Der Unterstrich dazwischen darf nicht fehlen. Aus diesem Feld werden Pfandangaben und Download-URLs erstellt. Außerdem spielt der Typ beim Scheduling eine wichtige Rolle (siehe auch Abschnitte 3.3.2 und 3.3.5ff).
- *platform:* Hier ist einzutragen, auf welcher Zielplattform der Job laufen muss/kann/darf/soll. Die Syntax lautet: <Software>.<Hauptspeicher>.<Upload-Datenmenge>.<CPU-Architektur>. Dabei hat man für die Softwarebeschreibung die folgenden Möglichkeiten: W für Windows (nativ), L für Linux (nativ), N für .NET (auf Windows), M für Mono (auf Linux), J für Java, P für Perl, Y für Python, R für die Skriptsprache R und G für das Algebrasystem GAMS. Benötigt ein Job eine Kombination dieser Plattformen, so sind die einzelnen Zeichen in beliebiger Reihenfolge aneinanderzuhängen. Für einen Job, der nur auf Windows mit Java läuft, müsste „WJ“ eingetragen werden. Ist ein Job auf mehreren Plattformen oder Plattformkombinationen lauffähig, trennt man die einzelnen Plattformbeschreibungen mit einem Semikolon. Beispielsweise „WJ;LJP“ für einen Job, der entweder Windows mit Java braucht, oder aber alternativ auf Linux mit Java und Perl lauffähig ist. Daran werden mit einem Punkt getrennt noch die Hauptspeicheranforderungen, die Ergebnisdatenmenge, die der Job erzeugt, sowie die benötigte Prozessorarchitektur angehängt. Sind diese Werte zum Entwicklungszeitpunkt nicht bekannt oder spielen sie keine Rolle, kann ein ‚*‘ eingetragen werden. Ein vollständiges Beispiel einer Plattformbeschreibung wäre „WN.*.100.*“ für einen Job, der unter Windows mit .NET läuft, keine Ansprüche an den installierten Hauptspeicher stellt, 100 MB Ergebnisdaten produziert und unabhängig von der Prozessorarchitektur arbeitet. Bei .NET- und Java-Programmen, die nicht auf nativen vorkompilierten Code zurückgreifen, ist das grundsätzlich so.
- *command:* Hier steht das Kommando, das der Agent startet, wenn alle Eingabedateien heruntergeladen sind. Auch Scripte (z. B. .bat, .cmd oder .sh-Dateien) sind möglich. Es sollte dabei aber immer bedacht werden, dass das Jobkommando mit den gleichen Benutzerrechten läuft wie der Agent, der den Befehl startet. Aus Sicherheitsgründen werden das im Allgemeinen nur stark eingeschränkte Benutzerrechte sein (siehe auch Abschnitt 3.5.1). Die Nutzer müssen davon ausgehen, dass

ihre Befehle/Programme nur im Anwendungsverzeichnis Schreibrechte haben und sich das restliche System ausschließlich lesend benutzen läßt.

- *resultFiles:* Hier werden die Dateien $a_1,\ldots,a_m$ angegeben, die der Job erzeugt und die nach Ausführung vom Agenten an den Server zurückgeschickt werden sollen. Sollte es sich dabei um mehr als eine Datei handeln, so sind die einzelnen Dateinamen durch ein Semikolon zu trennen, z. B. „result1.log;result2.log;result3.log". Es ist hier auch erlaubt, Pfadangaben zu benutzen, also etwa „subdir\results.log". Dabei sind als Pfadtrennzeichen sowohl der Slash ‚/' als auch der Backslash ‚\' erlaubt, der Agent korrigiert diese Zeichen entsprechend der Plattform, auf der er läuft. Diese Pfade werden dem Server beim Upload der Ergebnisdateien mit übergeben, dort wird die gleiche Verzeichnisstruktur automatisch nachgebildet. Der Job muss allerdings selbst dafür sorgen, dass er dem Rechenknoten diese Unterverzeichnisse erzeugt, da der Agent dort nach den angegebenen Dateien sucht. Letztendlich ist bei den Ergebnisdateien auch die Angabe eines ‚*' möglich. Dann lädt der Agent am Ende des Jobs alle Dateien zum Server, die neu entstanden sind oder schon vorhanden waren, sich jedoch während des Laufes geändert haben. Auch in diesem Fall wird die gesamte Unterverzeichnisstruktur berücksichtigt.
- *maintainoutput:* Wird hier ein YES eingetragen, speichert der Server eine Kopie der Ergebnisdateien im Downloadbereich. Da der Server einen Job nur dann an einen Agenten ausliefert, wenn alle spezifizierten Eingabedateien auch physisch vorhanden sind, lässt sich mit diesem Mechanismus eine Reihenfolge bei der Jobverarbeitung erzwingen bzw. lassen sich Datenabhängigkeiten der Jobs untereinander modellieren (z. B. ein Job, der die Ergebnisse anderer Jobs weiterverarbeitet, aber natürlich erst dann gestartet werden darf, wenn alle anderen Ergebnisse vorhanden sind). Benötigt man diese Funktionalität nicht, trägt man NO ein.
- *files:* Hier werden die Eingabedateien $e_1,\ldots,e_n$ spezifiziert. Wie bei den Ergebnisdateien können hier mehrere Dateinamen angegeben werden, auch sie sind jeweils durch ein Semikolon zu trennen. Einer Eingabedatei kann optional ein sogenanntes Plattformpräfix vorangestellt werden, welches anzeigt, für welche Plattform die Datei spezifiziert ist. Das ist dann sinnvoll, wenn ein Job auf verschiedenen Plattformen läuft (also die Befehlszeile auf unterschiedlichen Plattformen ausführbar ist), die Eingabedateien jedoch trotz eventueller Namensgleichheit binär verschieden sind. Der Befehl *foo.exe* sei beispielsweise unter Windows und Linux ausführbar, weil die Datei *foo.exe* in der jeweils passenden Binärform vorliegt. Man will damit erreichen, dass der Job auf beiden Plattformen ausgeführt werden kann, insgesamt jedoch nur einmal ausgeführt wird, also nur einmal auf dem Server vorliegt. Dann stellt der Nutzer den jeweiligen Dateinamen das Plattformpräfix und einen Doppelpunkt voran, der Eintrag für das Beispiel lautet dann: „W:foo.exe;L:foo.exe". Die jeweiligen Dateien selbst müssen sich dann auf dem Server in einem dem Präfix entsprechenden Unterverzeichnis befinden, im Beispiel also in *\mbo\bench\W* und in *\mbo\bench\L*. Diese Verzeichnisse muss der Nutzer über die Eingabedateienschnittstelle selbst anlegen. Bei der Plattformbeschreibung des Jobs müssen dann beide Plattformen angegeben werden. Im Bei-

spiel würde dort dann „W;L“ stehen. Der Agent lädt natürlich nur die Dateiversion herunter, die zu seiner eigenen Plattform passt. Bei den eigentlichen Eingabedateinamen sind auch beliebige Angaben mit der Wildcard ‚*‘ (beispielsweise „foo.*;*.exe;config*.xml“) möglich. Der Server bildet dann die Dateibeschreibung bei Auslieferung des Jobs anhand der real vorhandenen Dateien bzw. Verzeichnisse selbst und sorgt damit dafür, dass der Agent alle vorhandenen und zur Plattform passenden Dateien herunterladen wird. Der im vorigen Absatz beschriebene Datenabhängigkeitsmechanismus funktioniert bei solchen Wildcard-Angaben allerdings nicht mehr, da der Server jetzt nicht mehr prüfen kann, ob alle angegebenen Dateien schon vorhanden sind.

- *mailnotification:* Trägt man hier eine Email-Adresse ein, schickt der Server eine Benachrichtigung an diese Adresse, sobald der Job fertig ist. Legt man darauf keinen Wert, kann man das Feld leer lassen oder NO eintragen.
- *periodicUpload:* Trägt man hier ein YES ein, lädt der Agent periodisch (im Schnitt alle 30 min) die entstandenen Ergebnisdateien zum Server. Dies ist sinnvoll, wenn man bei lange laufenden Jobs Zwischenresultate einsehen will. Legt man darauf keinen Wert, kann man das Feld leer lassen oder NO eintragen. Ein YES schaltet zusätzlich die redundante Jobausführung (vgl. 3.3.8) ab.
- *userIdentifier:* Unter dem hier eingetragenen Bezeichner wird auf dem Server nach Ende des Jobs eine Datei angelegt, die die Konsolen- und Fehlerausgaben des Jobs, sowie seinen Exit-Code enthält. Trägt man z. B. hier „job123“ ein, erhält man zusätzlich zu allen Ergebnisdateien die Datei „job123.ALL“. Diese Datei wird automatisch vom Agent erzeugt, Nutzer müssen sich darum nicht kümmern. Sie sollten lediglich dafür sorgen, dass sie die Dateinamen wieder einem Job zuordnen können, sich also selbst passende, eindeutige Bezeichnungen ausdenken. Sollte diese Funktionalität nicht benötigt werden, lässt man das Feld leer.
- *preUserIdentifiers:* Hier trägt man (wiederum mit Semikolon getrennt) die *userIdentifier* derjenigen Jobs ein, die zeitlich vor diesem Job erledigt sein müssen. Damit lässt sich ebenfalls eine Reihenfolge der Jobs erzwingen, ohne den Datensynchronisationsmechanismus verwenden zu müssen. Trägt man hier etwa „test1;test2“ ein, dann wird dieser Job erst dann gestartet werden, wenn die Jobs mit den *userIdentifier*n „test1“ und „test2“ fertig sind. Die Jobs lassen sich damit also als gerichteter azyklischer Graph modellieren. Spielt die Reihenfolge der Jobs keine Rolle, lässt man dieses Feld leer. Hat man Reihenfolgeabhängigkeiten definiert, kann man sich diese graphisch darstellen lassen (Abb. 3.23), wobei auch eventuell vorhandene Zyklen erkannt und optisch hervorgehoben werden.

Wichtig bei diesen 10 Einträgen ist, dass keinesfalls Tabulatorzeichen verwendet werden.

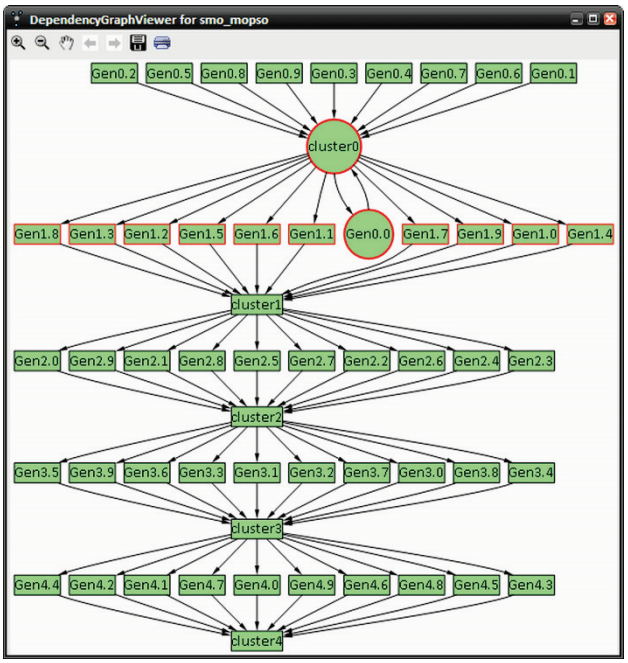


Abb. 3.23 Abhängigkeitsgraph mit erkanntem Zyklus

Die einzelnen Jobdaten kann man auch in einer Textdatei erzeugen und diese dann in den Jobmanager laden. Eine Jobdatei enthält in jeder Zeile die Beschreibung eines Jobs, die einzelnen Felder werden dort immer mit einem Tabulatorzeichen getrennt. Eventuell nicht benötigte Felder wie *maintainoutput* oder *mailnotification* füllt man auch hier am besten mit einem NO aus. Das Textfile kann man mit einem Klick auf … auswählen, ein Klick auf *Load File…* lädt die Datei in die Upload-Tabelle. Alternativ kann man die Jobdatei per Drag-and-Drop auf die Tabelle ziehen. Ein weiterer Klick auf *Upload* befördert die Jobs schließlich zum Server.

Bearbeitungszustand vorhandener Jobs

Mit den *Refresh*-Buttons im Reiter *View Jobs and Statistics* (Abb. 3.24) lässt sich der Zustand der Jobs auf dem Server anzeigen. Die Jobs werden dann in der oberen Tabelle dargestellt. Mit dem *JobType Filter* und den Optionen *show DONE*, *show FREE* etc. kann man die Sicht auf die Datenbasis entsprechend einschränken. Die Daten in den beiden Tabellen lassen sich durch einen Klick auf die entsprechende Spalte sortieren. Das Tool zeigt hier auch ein Diagramm, welches den zeitlichen Verlauf der Jobausführung über den Zeitraum von fünf Stunden graphisch darstellt.

Sind Jobs fertig (*status* DONE), holt man sich über die Ergebnisdateien-Schnittstelle deren Resultate ab. Neben den selbst spezifizierten Ergebnisdateien ist dort (falls gewünscht) auch die durch den *userIdentifier* angegebene Ausgabeumleitungsdatei erhältlich.

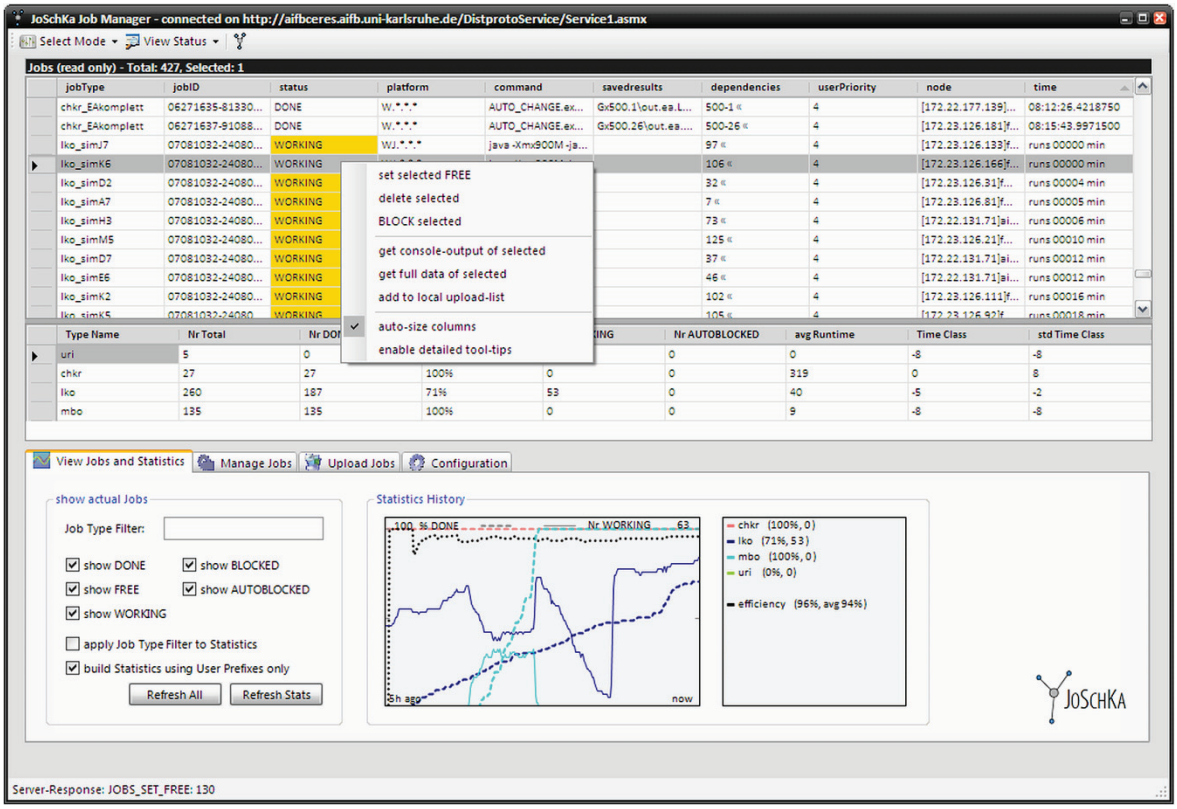


Abb. 3.24 Jobmanagement-Tool: Jobdaten

Vorhandene Jobs verwalten

Der Reiter *Manage Jobs* (Abb. 3.25) bietet verschiedene Funktionen zur Verwaltung der auf dem Server verfügbaren Jobs. Man markiert die betreffenden Zeilen in der oberen Tabelle und hat dann die Möglichkeit

- diese Jobs freizugeben (um einen blockierten oder automatisch blockierten Job wieder zur Berechnung bereitzustellen),
- diese Jobs zu löschen oder deren Ausführung zu blockieren,
- die maximale Anzahl an fehlgeschlagenen Rechenläufen zu ändern (ein Job, der öfter als die hier eingestellte Anzahl fehlschlägt, wird automatisch vom System blockiert),
- die maximale Joblaufzeit (in Stunden) zu ändern (−1 steht für unendlich) und
- und die benutzerdefinierte Priorität zu ändern (je höher der Wert, desto höher die Priorität, höher priorisierte Jobs werden vom Scheduler bevorzugt, vgl. auch Abschnitt 3.3.5), sowie
- die Ausführung eines Jobs für einen bestimmten Rechner zu reservieren.

Die nicht parametrisierbaren Funktionen stehen auch per Kontextmenü zur Verfügung. Dieses Menü, das sich öffnet, wenn man einen Rechtsklick auf einen Job tätigt, bietet noch weitere Funktionen: die Ansicht der Ausgabeumleitung und umfassende Informationen über seinen Zustand in der Datenbank des Servers, die nicht in der Tabelle ange-

zeigt werden (*get console-output of selected* und *get full data of selected*). Weiterhin kann man einen Job als Vorlage verwenden, um neue Jobs zu generieren (*add to local upload list*). Schließlich kann man mit dem (hier nicht gezeigten) Reiter *Configuration* die URL zum Server ändern, Benutzerdaten wie Login und Passwort angeben und die Schriftarten der Tabellendarstellung festlegen.

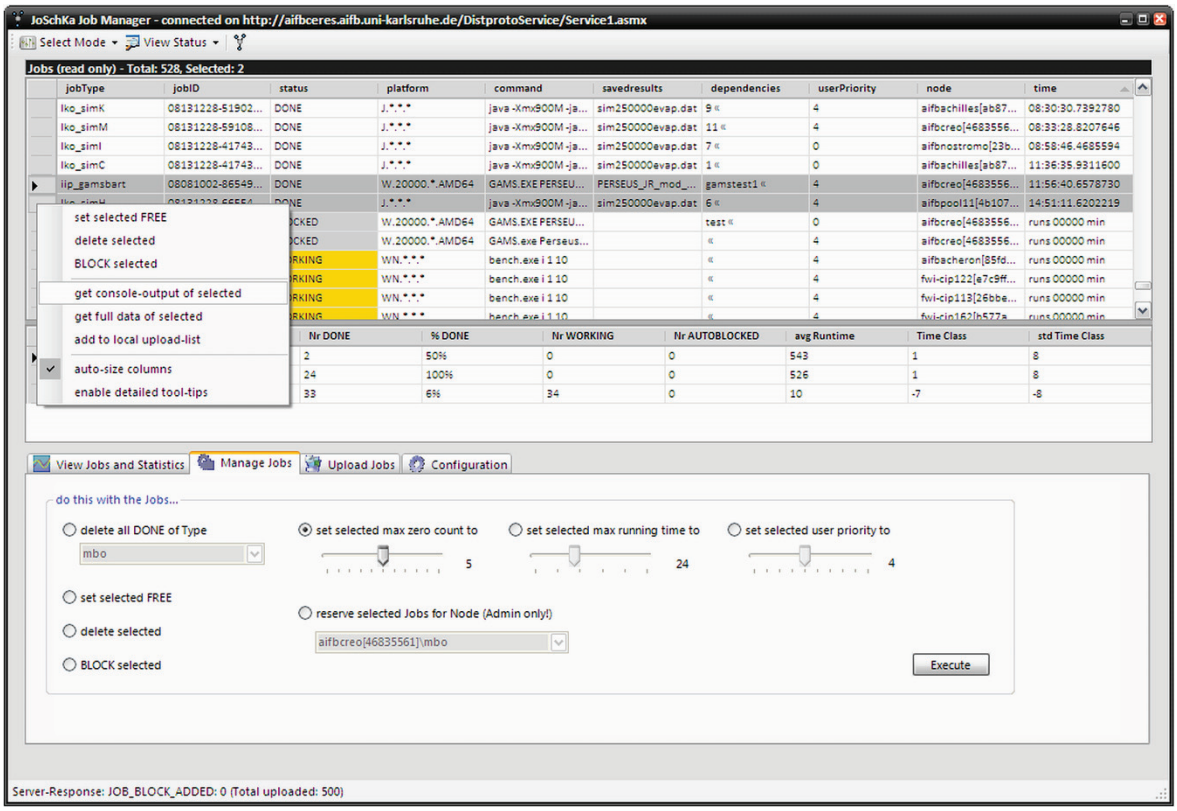


Abb. 3.25 Jobmanagement-Tool: Jobverwaltung

Weitere Funktionen

Mit einem Klick auf die Buttons *Agents…*, *Nodes…* und *WebMethods…* in der *View Status*-Menüleiste öffnen sich weitere Fenster, mit denen man sich unter anderem einen Überblick über die am System bekannten Rechenknoten verschaffen kann.

Das Fenster *Nodes* (siehe Abb. 3.26) gibt einen Überblick darüber, welche Daten der JoSchKa-Server von jedem Rechenknoten erfasst hat. Optisch besonders hervorgehoben werden dabei die Daten, die bei den einzelnen Verteilstrategien eine besondere Rolle spielen (siehe Abschnitte 3.3.4ff). Dargestellt werden auch die Verteilung der Zuverlässigkeitswerte und die Verteilung der durchschnittlichen Arbeits- und Verfügbarkeitszeiten jedes Rechners, wobei letztere Graphik linear und logarithmisch skaliert werden kann.

Das (hier nicht gezeigte) Fenster *Agents* dient dabei dazu, einzelne Agenten auf bestimmte Jobtypen oder Plattformen festzulegen, sofern das nötig ist. Dies ist jedoch nur zu administrativen Zwecken gedacht und dem normalen Nutzer nicht erlaubt. Das (ebenfalls nicht gezeigte) Fenster *WebMethods* gibt Auskunft über die Häufigkeit der Webmethodenaufrufe am Server, sowie deren Ausführungszeit, es dient nur zu statistischen Zwe-

cken und Performancemessungen. Für den normalen Benutzer haben diese Fenster keine Bedeutung.

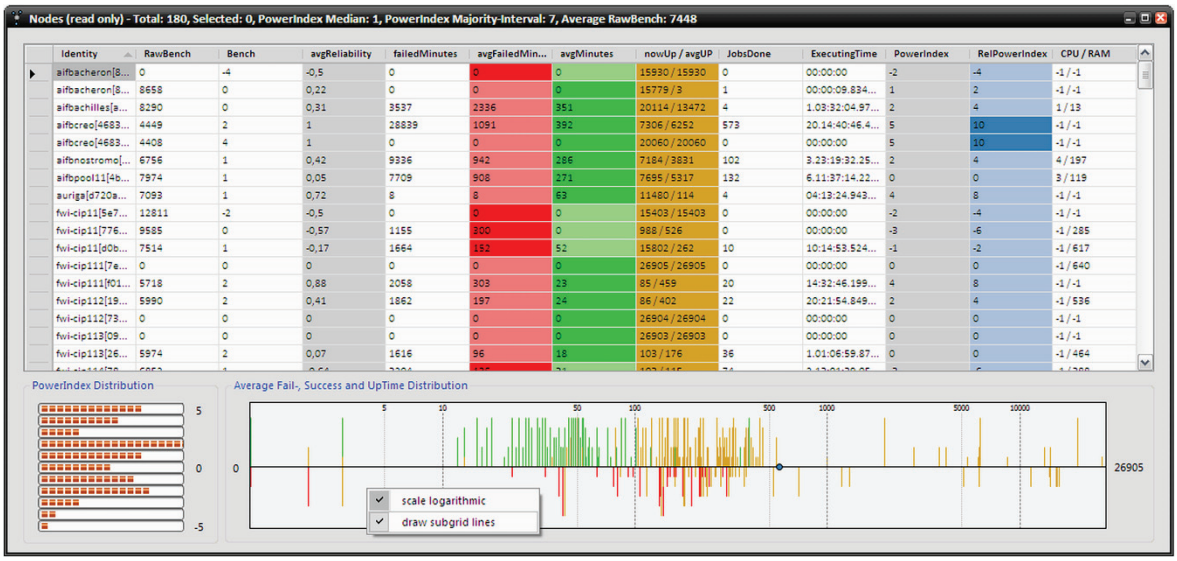


Abb. 3.26 Übersicht über Rechenknoten und deren Leistung

3.7 Die JoSchKa Thread-API für .NET

Wie aus dem vorherigen Abschnitt ersichtlich, besteht der typische Arbeitsablauf beim Benutzen von JoSchKa darin, zunächst das auf den Rechenknoten auszuführende Programm zu implementieren, zu kompilieren und über die SMB-Schnittstelle auf dem Server zu hinterlegen. Danach definiert der Nutzer mit Hilfe des Management-GUIs die einzelnen Jobs, lässt sie starten und holt sich zu einem späteren Zeitpunkt die erstellten Ergebnisdateien wiederum per SMB ab. Dies zeigt, dass die Benutzung von JoSchKa in seiner bisherigen Form nur eingeschränkt interaktiv möglich ist:

- Der Ablauf der parallelen Anwendung und eventuelle Reihenfolgeabhängigkeiten müssen vorab fix definiert werden.
- Die automatische Reaktion auf das Vorhandensein von Ergebnissen ist nur eingeschränkt oder umständlich möglich.

Aus diesem Grund wurde die JoSchKa Thread-API entwickelt, eine Serviceschicht, mit der man das Verteilsystem programmgesteuert nutzen kann. Hierzu wurden die Schnittstellen des Servers um Funktionen erweitert, die den programmgesteuerten Upload von ausführbarem Code, sowie das Abholen seiner Ergebnisse ermöglichen. Zusätzlich entstand eine clientseitig zu nutzende Klassenbibliothek, die folgende Funktionen kapselt:

- Erzeugen von Jobs (Threads)
- Upload beliebiger lokaler Dateien
- Über das Vorhandensein der Ergebnisse informieren
- Abholen der Ergebnisse

Der Nutzer der API muss lediglich einen leeren Methodenrumpf mit dem parallel auszuführenden Programmteil implementieren, der Code selbst wird in Form einer kompilierten .NET-Assembly (in Form einer DLL) zum Server und von dort zum Rechenknoten übertragen. Die folgenden beiden Abbildungen (Abb. 3.27, Abb. 3.28) zeigen die Klasse *DistProtoAPIWorker*, deren *Execute()*-Methode vom Nutzer zu implementieren ist, sowie die Klassenstruktur der Programmierschnittstelle.

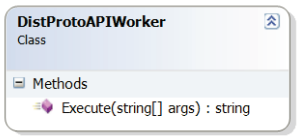


Abb. 3.27 Klassenrumpf für mobilen parallelen Code

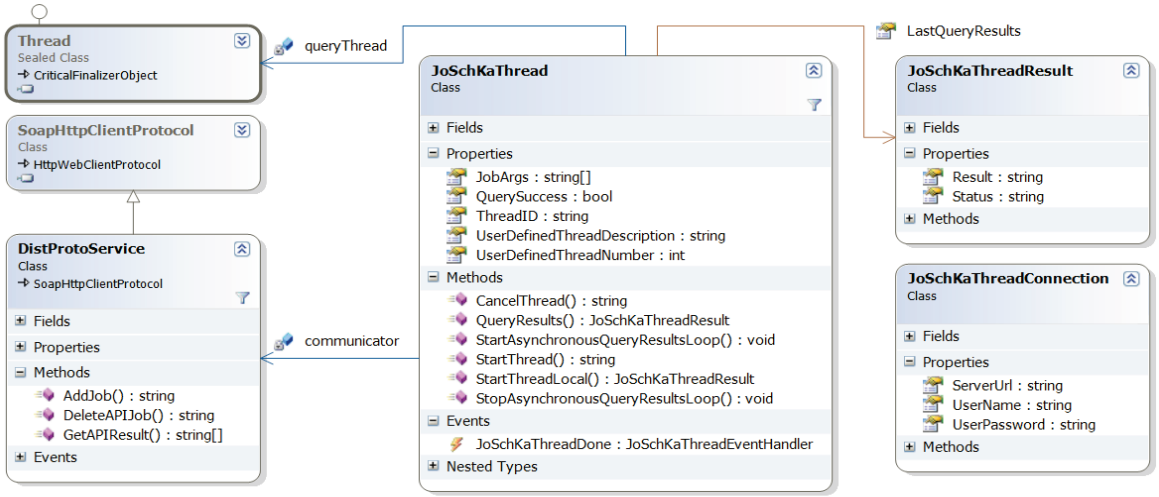


Abb. 3.28 Die JoSchKa Thread-API

Der Kern der API wird von der Klasse *JoSchKaThread* gebildet. Sie lässt sich wie die gängigen Thread-APIs benutzen: Man instanziiert einen Thread, parametrisiert (*JobArgs*), konfiguriert (*UserDefinedThreadDescription*, *UserDefinedThreadNumber*) und startet ihn (*StartThread()*). Während der Ausführung besteht die Möglichkeit, nach Ergebnissen zu fragen (*QueryResults()*, *LastQueryResults*) oder sich automatisch durch ein sogenanntes Event benachrichtigen zu lassen, sobald der Thread beendet ist und ein Ergebnis vorliegt (*JoSchKaThreadDone*). Die Klassen bieten auch die Möglichkeit, einen Thread vorzeitig abzubrechen (*CancelThread()*). Beim Instanziieren eines Threads benötigt man ein Objekt der Klasse *JoSchKaThreadConnection*, das die Zugangsdaten zum Server speichert, die Ergebnisse werden durch eine Instanz der Klasse *JoSchKaThreadResult* repräsentiert. Der SOAP-Kommunikationsproxy *DistProtoService* ist bereits in die API integriert.

Das folgende Ablaufdiagramm (Abb. 3.29) soll die Arbeitsweise und die Nutzung der Bibliothek anhand eines einfachen Beispiels erläutern. Angenommen, man hat die *Execute()*-Methode der *DistProtoAPIWorker*-Klasse so implementiert, dass das Produkt zweier Zahlen berechnet wird. Zunächst wird der Thread erzeugt und parametrisiert. Nach seinem Start wird die lokale DLL, die den Multiplikationscode enthält, vom Thread-Objekt über

eine spezielle Schnittstelle zum Server übertragen und von diesem in das Quelldatenverzeichnis des Entwicklers abgelegt. Zusätzlich kopiert der Server die Datei *DistProtoAPIExecutor.exe* in das Quellverzeichnis. Dieses Programm übergibt später beim Rechenknoten die Threadparameter zur Berechnung an die DLL und schreibt deren Ergebnisse in eine Datei. Anschließend erzeugt die API über einen SOAP-Aufruf den eigentlichen JoSchKa-Job. Dieser wird dann wie in Abschnitt 3.4 beschrieben ausgeführt, nach Beendigung wird die Ergebnisdatei zum Server übertragen. Der Nutzer kann dann den Inhalt der Ergebnisdatei über die API einsehen oder sich asynchron über einen sogenannten .NET-Event darüber benachrichtigen lassen.

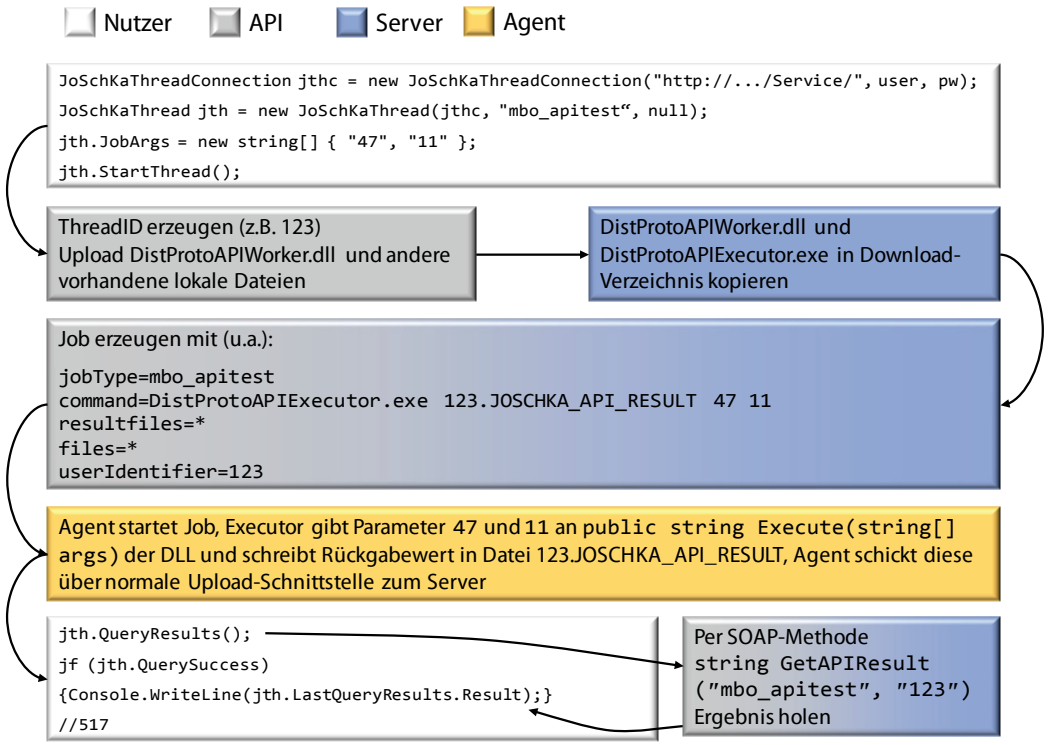


Abb. 3.29 Nutzung und Arbeitsweise der API

Der Vorteil bei der Benutzung dieser Entwicklerschnittstelle für den Nutzer liegt darin, dass es damit möglich ist, flexibel auf die Ergebnisse von Threads zu reagieren und gegebenenfalls neue Threads zu erzeugen bzw. vorhandene abzubrechen, ohne sich über die Arbeitsweise von JoSchKa Gedanken machen zu müssen. Die API versteckt diese Interna vor dem Nutzer vollständig.

3.8 Einordnung und Vergleich

JoSchKa wurde unter dem Gesichtspunkt entworfen, ein System zur Verteilung von CPU-intensiven Jobs auf Standard-PC zu realisieren. Aus diesem Grund wurde ein ähnlicher Architekturansatz wie bei BOINC und Alchemi gewählt: Es handelt sich um eine Client/Server Architektur, bei der Standard-Internetprotokolle und Datenformate eingesetzt werden. JoSchKa verhält sich gerade im Vergleich zu BOINC generischer was die Art der Jobs, die unterstützten Programmiersprachen und Laufzeitumgebungen angeht

und ist für Entwickler paralleler Anwendungen ähnlich schnell ad hoc nutzbar wie beispielsweise ein Condor-Cluster.

Das Arbeitsprinzip, mit dem JoSchKa die Jobs verteilt, hat gewisse Ähnlichkeit mit dem Tuplespace-Ansatz [Gel85], da die Rechenknoten selbsttätig entsprechend dem pull-Prinzip Jobs vom Server anfordern. Dies und die Wahl des HTTP-Protokolls in Kombination mit Webservices macht das System zwar im Internet einsetzbar, bringt jedoch auch gewisse Nachteile mit sich. Zum einen kann der Server einem Agenten nicht gezielt Steuerdaten zuschicken, sondern ist immer darauf angewiesen, dass der Agent sich von alleine meldet und sich entsprechende Anweisungen abholt. Diese Abfragefrequenz bestimmt, wie schnell das System z. B. beginnt, neue Jobs an nicht-beschäftigte Knoten auszuliefern. Je häufiger ein Agent nach Jobs fragt, desto schneller wird mit der Bearbeitung dieser neuen Jobs begonnen, der Server wird jedoch durch die gehäuften Anfragen leichter überlastet. Umgekehrt verhält es sich genauso: Je sparsamer die Agenten Anfragen an den Server stellen, desto träger wird das System, der Server kann nun allerdings mehr Agenten bedienen, bevor er überlastet wird. Da die SOAP-Kodierung der übertragenen Steuerdaten einen gewissen Kommunikationsoverhead mit sich bringt und die notwendige XML-Verarbeitung am Server CPU-Ressourcen benötigt, wurden einige Maßnahmen ergriffen, um die Netzwerk- und Hardwarebelastung des Servers zu verringern (vgl. Abschnitt 4.3).

Eine wesentliche Eigenschaft von JoSchKa besteht im Monitoring der unterschiedlichen Rechner, auf die es die Jobs verteilt, und in der Anpassung der Jobverteilung an eben diese Beobachtungen. Die Heterogenität der Rechenknoten beschränkt sich nämlich leider nicht nur auf die Hard- und Softwareausstattung der Rechner, sondern auf deren Verfügbarkeit. Man kann grundsätzlich nicht davon ausgehen, dass ein Rechner einen Job zuverlässig bis zum Ende durchführt, sondern muss – gerade im Pool- und Arbeitsplatzbetrieb – mit permanenten Neustarts der PCs rechnen. Diese Ausfallhäufigkeit kann sich aber von Rechner zu Rechner individuell unterscheiden, sie kann regelmäßig oder aber völlig chaotisch erfolgen. Stehen dem JoSchKa-System mehrere verschiedenartige Jobs zur Verfügung, die sich in ihren Laufzeitanforderungen geeignet unterscheiden, werden diese durch Einsatz spezieller Algorithmen so verteilt, dass die durch Rechnerausfälle unnütz verbrauchte Rechenkapazität verringert wird. Das geschieht allerding immer unter Berücksichtigung der Bedingung, allen Benutzern möglichst die gleiche Anzahl an Rechenressourcen zur Verfügung zu stellen und, als wichtigste Randbedingung, Benutzer mit gleichen Anforderungen auch gleich zu behandeln.

Selbstverständlich kann das Zuverlässigkeitsverhalten eines einzelnen Rechners durch den Schedulingalgorithmus nicht beeinflusst werden, es ist jedoch durchaus möglich, die jeweiligen Gegebenheiten so zu nutzen, dass die Leistung des Gesamtsystems gesteigert wird. Eine wichtige Voraussetzung dazu ist allerdings ein gutmütiges (sprich: vorhersagbares) Verfügbarkeitsmuster der Rechner. Derartige Beobachtungsmechanismen und die Fähigkeit, die Verteilung der Jobs dem gemessenen Verhalten der Rechenknoten – mit dem Ziel der Leistungssteigerung – anzupassen, finden sich bei keinem der im Rahmen dieser Arbeit vorgestellten und mit JoSchKa vergleichbaren Verteilsysteme.

Zusätzlich wurde ein anwenderfreundliches graphisches Tool zum Erstellen und Verwalten der Jobs realisiert, was JoSchKa von anderen rein kommandozeilenorientierten Systemen unterscheidet. Zur programmgesteuerten Nutzung wurde eine .NET-basierte Programmierschnittstelle entwickelt, die die normale Nutzung durch klassische Batchjobs um die Möglichkeit einer transparenten parallelen Verteilung von objektorientiertem Programmcode nach dem Master-Worker-Prinzip erweitert.

Kapitel 4

Implementierung und Betrieb

Dieser Teil der Arbeit gibt zunächst einen Einblick in die interne Strukturierung des Programmcodes der Komponenten von JoSchKa. Anschließend wird detailliert die Installation und der Betrieb der beiden Systemkomponenten beschrieben, wobei sich das Kapitel auf die notwendigen administrativen Arbeiten konzentriert. Abschließend wird der Einfluss von Hintergrundjobs auf die reguläre Nutzung von Arbeitsplatzrechnern untersucht.

Bei der Implementierung der einzelnen Bestandteile von JoSchKa wurde die Sprache C# verwendet, das System setzt auf Microsofts .NET-Framework (in der Version 2.0) auf. Der Server benötigt einen ASP.NET Application Server, der innerhalb eines MS Internet Information Server IIS Version 6 läuft [Sch06b]. Alle anderen Komponenten, wie z. B. das Management-GUI, benötigen ebenfalls das .NET-Framework 2.0, die Agenten ohne graphische Benutzeroberfläche sind auch unter Linux/Unix funktionsfähig, sofern dort eine aktuelle Mono-Laufzeitumgebung existiert.

4.1 Server

Bei der Serverkomponente handelt es sich um eine normale dynamische Webanwendung, die um eine Webservice-Schnittstelle erweitert wurde. Im Gegensatz zu klassischen Webtechnologien wie PHP (PHP Hypertext Preprocessor) oder SSI (Server Side Includes), die nur auf konkrete Requests in Aktion treten und antworten, ist es mit ASP.NET möglich, interne Programmabläufe zu realisieren, die als Thread ohne Benutzeranfrage permanent im Hintergrund laufen und so den Datenbestand überwachen. Weiterhin ist es möglich, auf globale Ereignisse wie das Starten oder Beenden des Webdienstes zu reagieren, womit es möglich wird, eigentlich persistent zu speichernde Daten komplett im Hauptspeicher des Rechners zu halten, was die Performanz stark steigert. Neben der .asmx-Webservice-Schnittstelle und einigen .aspx-Seiten, die der Statusausgabe als HTML-Webseite dienen, besitzt die Serveranwendung einige weitere Klassen (Abb. 4.1), die im Folgenden erläutert werden.

Der Aufruf der Webmethoden wird von ASP.NET an die SOAP-Schnittstellenklasse (*Service1*) delegiert. Die meisten Aufrufe werden von dort jedoch an spezialisierte Klassen weitergereicht. Die zentrale Rolle spielt die Datenverwaltungsklasse (*DataManager*), die das gesamte Jobportfolio verwaltet.

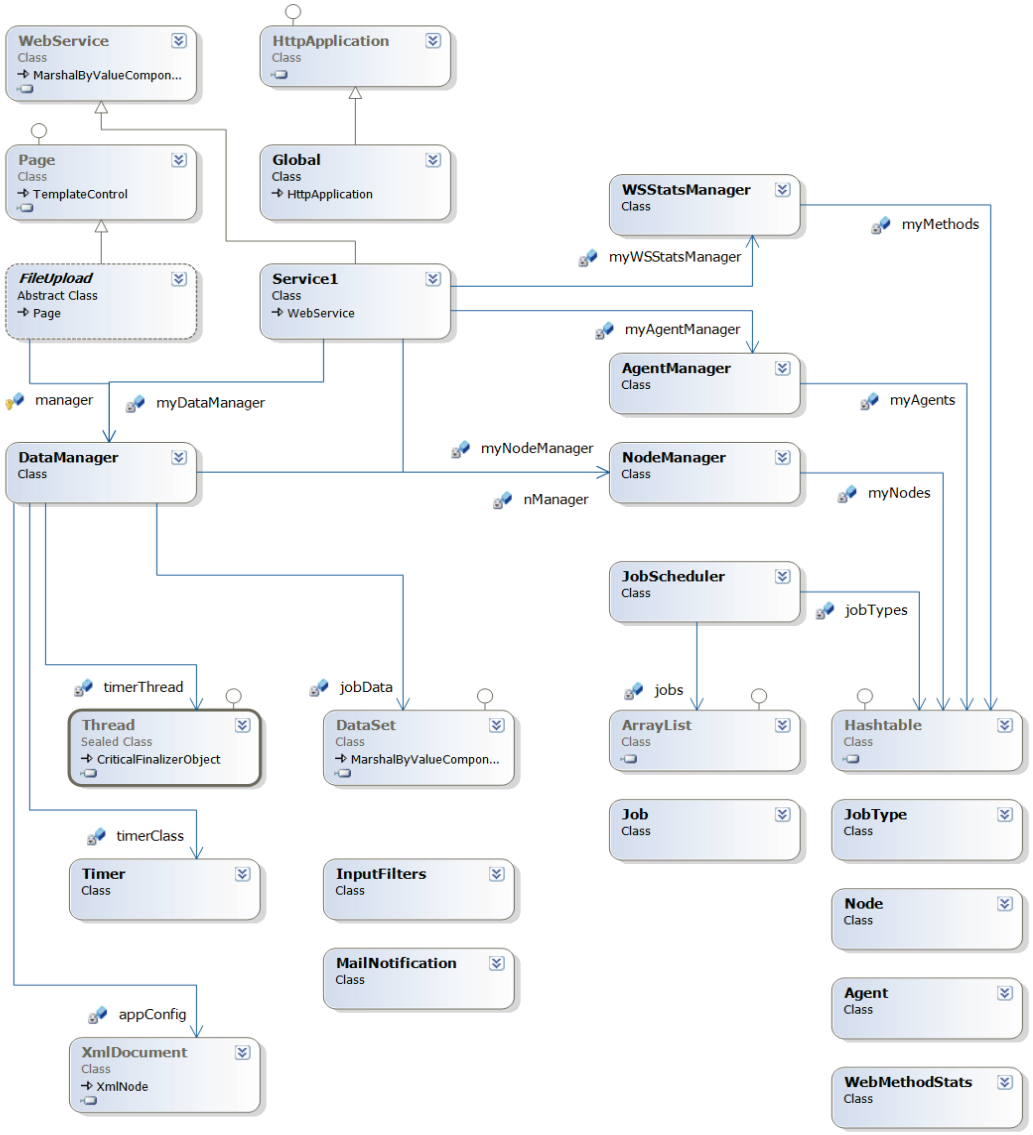


Abb. 4.1 Klassendiagramm Server

Neben Methoden zum Auswählen eines Jobs existieren hier auch Routinen zum Verwalten (Hinzufügen, Löschen etc.) derselben. Die Klassen *JobScheduler*, *JobType* und *Job* werden beim Scheduling benötigt, sie speichern statistische Daten über die verschiedenen Jobtypen und werden benutzt, einen konkreten Job auszuwählen, sobald ein Agent eine Anfrage danach stellt (vgl. Kapitel 3.3.5ff). Sofern ein passender Job gefunden wurde, werden der Name und die IP-Adresse des Agenten (bzw. des Rechners, auf dem er läuft) an die Agentenverwaltung (*AgentManager*) gemeldet, dort wird der Agent als aktiv vermerkt. Diese Agentendaten können ebenfalls abgefragt werden, so dass man immer einen Überblick darüber hat, wie viele davon dem System bekannt sind und was sie jeweils tun.

Die in Abschnitt 3.3.4 beschriebenen Leistungswerte der einzelnen Knoten werden von der Klasse *NodeManager* gesammelt und verwaltet. Weiterhin existiert eine Klasse, die die Implementierung des Hintergrund-Threads für die Countdown-Verwaltung enthält (*Timer*), eine weitere, die den Versand von Benachrichtigungsmails realisiert (*MailNotification*), sowie eine Klasse, die die Dateiupload-Schnittstelle implementiert (*FileUpload*). Methoden der Klasse *InputFilters* prüfen Benutzereingaben auf Plausibilität und

filtern sicherheitskritische Zeichen weg. Zwei weitere Komponenten erzeugen HTML-Seiten, mit denen man sich browserbasiert einen Überblick über den Zustand der Jobdaten und der Agenten bzw. Rechenknoten verschaffen kann (*JobStatus.aspx*, *AgentStatus.aspx* und *NodeStatus.aspx*, im Diagramm nicht gezeigt). Zusätzlich existieren noch einige Hilfsklassen, die teilweise reinen Debugzwecken dienen (*WSStatsManager* z. B. dient dem Sammeln von statistischen Informationen über die einzelnen Webmethoden). Die Klasse *Global* steht jeder ASP.NET-Webanwendung zur Verfügung und enthält Methoden, die bei verschiedenen globalen Ereignissen (wie etwa Start und Ende der Webanwendung) aufgerufen werden. Durch Implementieren der dort verfügbaren Methodenrümpfe kann z. B. auf das Herunterfahren des Servers reagiert werden.

Die Klassen *DataManager*, *NodeManager* und *AgentManager* verwalten zentrale Datenstrukturen, die aus Performanzgründen permanent im Hauptspeicher gehalten werden. Damit man innerhalb der gesamten Anwendung auf die gleichen Datenstrukturen zugreifen kann, wurden diese Klassen als Singleton realisiert, parallele Zugriffe darauf werden durch Threadsperren synchronisiert. Die eigentlichen Daten werden über Tabellen und andere dynamische Datenstrukturen wie Listen und Hashtabellen verwaltet und als XML-Dokument persistent gespeichert (Listing 4.1). Ein XML-Element zur Speicherung eines einzelnen Jobs hat die folgende Struktur (siehe auch Abschnitt 3.3.2):

Listing 4.1 Gespeicherte Jobdaten

```
<Table1>
     <jobID>0123456789987</jobID>
     <jobType>mbo_jaws</jobType>
     <status>DONE</status>
     <countdown>3</countdown>
     <platform>WN</platform>
     <command>jaws.exe -l=10 -t=50</command>
     <resultfiles>*</resultfiles>
     <savedresults>results.log;error.log</savedresults>
     <maintainoutput>NO</maintainoutput>
     <files>jaws.exe</files>
     <node>[127.0.0.1]AIFBACHERON\Bonn.1</node>
     <time>0:10:3.1976</time>
     <mailnotification>mbo@aifb.uni-karlsruhe.de</mailnotification>
     <nrzerocountdowns>0</nrzerocountdowns>
     <maxzerocountdowns>7</maxzerocountdowns>
     <maxrunningtime>34</maxrunningtime>
     <rndID>3911</rndID>
     <tempupload>NO</tempupload>
     <userIdentifier>jaws.l10.t50</userIdentifier>
     <userPriority>9</userPriority>
     <preUserIdentifiers></preUserIdentifiers>
     <commitTimestamp>633537880671776544</commitTimestamp>
</Table1>
```

Die im Dateisystem gespeicherte Datei enthält für jeden Job ein solches <*Table1*> Element. Beim Start des Servers wird diese Datei deserialisiert und steht dann wieder für die interne Verarbeitung als Tabelle zur Verfügung. Die von *NodeManager* gespeicherten Da-

ten über einen Rechenknoten (siehe auch Abschnitt 3.3.4) werden intern ebenfalls über objektorientierte Datenstrukturen verwaltet, persistent gespeichert werden sie jedoch ebenfalls als XML-Dokument und folgen der folgenden Struktur (Listing 4.2):

Listing 4.2 Gespeicherte Knotendaten

```
<node identity="[172.22.131.117]aifbachilles\mbo">
     <time>22.09.2006 22:12:51</time>
     <timestamp>1055502122</timestamp>
     <rawbench>6038</rawbench>
     <bench>2</bench>
     <totaljobscomitted>5</totaljobscomitted>
     <executingtime>16.03:25:39.5855250</executingtime>
     <cpuqueue>2</cpuqueue>
     <freeram>70</freeram>
     <totalfailedminutes>288</totalfailedminutes>
     <timequeue>1368,1596,10725,4418,5138</timequeue>
     <failtimequeue>105,97,86</failtimequeue>
     <uptimequeue>127,214,342,987</uptimequeue>
     <successerrorhistory>1,1,-1,-1,1,-1,1,1</successerrorhistory>
</node>
```

4.2 Agent

Das Agentenprogramm wurde so konzipiert, dass es ohne jegliche Benutzerinteraktion auskommt. Zur Steuerung benötigt man lediglich eine Konfigurationsdatei und/oder Kommandozeilenparameter. Zur Laufzeit lässt sich der Agent über einige Serverfunktionen beeinflussen. Das Programm existiert in einer Version ohne graphische Benutzeroberfläche und kann somit als Dienst bereits beim Systemstart im Hintergrund gestartet werden. Unter Windows funktioniert das mit dem *Taskplaner* (siehe 4.5.2), unter Linux richtet man einen sogenannten *Cronjob* ein, der dafür sorgt, dass der Agent beim Booten des Systems mit gestartet wird. Eine zweite Version der Anwendung wurde mit einer graphischen Oberfläche ausgestattet. Sie kann als Bildschirmschoner betrieben werden, so dass ein Arbeitsplatzrechner wirklich nur dann Jobs berechnet, wenn der Benutzer den Rechner nicht verwendet. Ein weiteres Feature der graphischen Version ist das Einfrieren eines Jobs: Bei Interaktion des lokalen Benutzers mit der Maus oder der Tastatur kann ein laufender Job komplett angehalten und später wieder fortgesetzt werden.

Wie aus dem Diagramm (Abb. 4.2) ersichtlich wird, ist das Programm in 7 Klassen unterteilt. *Agent* stellt das Hauptprogramm dar: Sie liest die Konfiguration ein, verwaltet die einzelnen Verzeichnisse und ruft in der Hauptschleife die einzelnen Methoden der Managerklasse (*JobManager*) auf. Diese Klasse kapselt die einzelnen Vorgänge, die beim Herunterladen von Jobs und Dateien, dem Ausführen, dem Hochladen von Ergebnisdateien und bei der Ausführungsbestätigung ablaufen, und speichert die aktuelle Jobspezifikation in *Job*. *FSLock* bietet statische Methoden zum Sperren des Cacheverzeichnisses auf Dateisystemebene, in *LifeLine* laufen die beiden von *JobManager* instanziierten Threads, die während der Ausführung Statusmeldungen mit dem Server austauschen, die Systemlast messen und gegebenenfalls den Jobabbruch veranlassen.

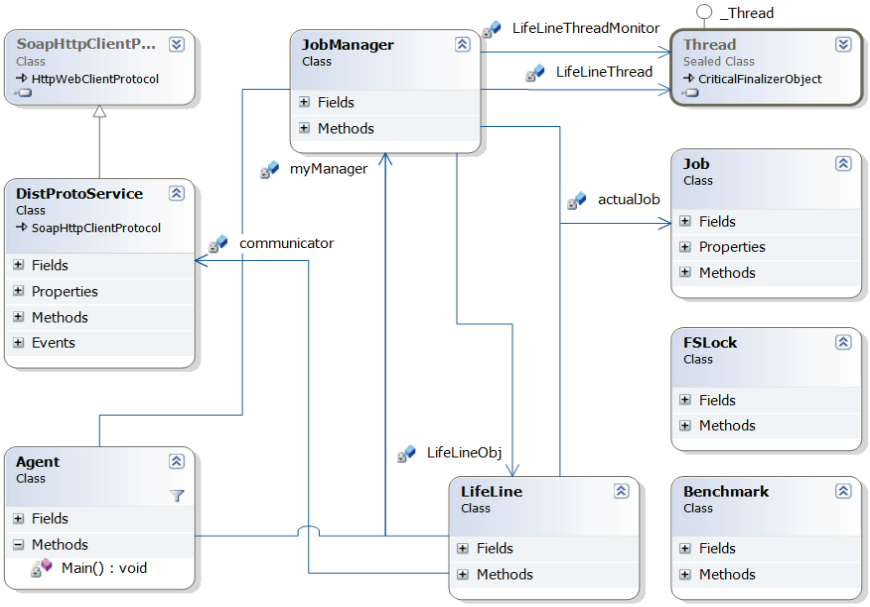


Abb. 4.2 Klassendiagramm Agent

Die Klasse *Benchmark* ermöglicht einen Test, der die Systemleistung im arithmetisch-logischen Bereich testet und die Netzwerkbandbreite zum Server bestimmt. Der arithmetische Test dauert auf einem 3 GHz-System etwa 20 Sekunden. Die Klasse *DistProtoService* kapselt die Webservice-Zugriffe auf den JoSchKa-Server. Sie wird aus der WSDL-Beschreibung des Webservice automatisch generiert.

Das Agenten-Programm wurde in C# implementiert und läuft unter Windows, benötigt dort die .NET-Laufzeitumgebung in der Version 2.0. Das gleiche Binary läuft ohne neu kompiliert werden zu müssen unter Unix/Linux, wenn dort die Mono-Laufzeitumgebung installiert ist. Es muss lediglich mit dem Befehl *chmod u+x DistProtoAgent.exe* ausführbar gemacht werden.

Die Implementierung enthält intern einige betriebssystemabhängige Teile, in die, je nachdem, welche Plattform zum Einsatz kommt, automatisch richtig verzweigt wird:

- Das Auslesen der Systemlast und des freien Hauptspeichers werden unter Windows unter Verwendung einiger Systembibliotheken realisiert, während man unter Linux die Dateien */proc/meminfo* bzw. */proc/cpuinfo* lesen und parsen muss.
- Der Jobabbruch musste unterschiedlich realisiert werden, denn das .NET-Framework kann nur Prozesse, aber keine Prozessstrukturen (Prozesse einschließlich deren gestarteter Subprozesse) abbrechen, und so wurde auf die Systembefehle *taskkill /T /F* bzw. *kill -9* zurückgegriffen.

Weil das .NET-Framework und die Mono-Laufzeitumgebung diese systemnahen Routinen nicht in der plattformunabhängigen Klassenbibliothek kapseln, kann auf diese Verzweigungen und die Implementierung einiger systemabhängiger Routinen nicht verzichtet werden. In zukünftigen Versionen dieser Laufzeitumgebungen könnte sich das aber noch ändern, so dass das Programm nicht nur nach außen hin, sondern auch intern vollständig unabhängig vom verwendeten Betriebssystem ist.

4.3 Performanz und Maßnahmen zur Leistungssteigerung

Um die Leistungsfähigkeit des Systems auch bei großen Jobmengen und einer großen Anzahl von Rechenknoten gewährleisten zu können, wurden einige Maßnahmen ergriffen, die den Server, der in einem Client/Server-System den zentralen Angriffs- und Ausfallspunkt bildet, so leistungsfähig wie möglich machen. Neben einer effizienten Implementierung mit schnellen internen Datenstrukturen (siehe Abschnitt 4.3.1) und Algorithmen wurden die in den danach folgenden drei Abschnitten beschriebenen zentralen Mechanismen zur Performance-Verbesserung realisiert[10].

4.3.1 Interne Datenstrukturen

Wie in Abschnitt 4.1 erläutert, verwaltet der Server die Jobs in einer einzelnen Tabelle, bei der jede Zeile alle Informationen über einen Job speichert. Bei den einzelnen auf diesen Daten ausgeführten Operationen ist es jedoch sinnvoll, zusätzliche Datenstrukturen zu verwenden, die die anfallenden Routinen bei der Jobverwaltung stark beschleunigen.

Indextabellen

Eine permanent durchzuführende Prozedur stellt das Auffinden eines konkreten Jobs in der Jobtabelle dar, z. B. weil der Benutzer genau diesen Job löschen möchte, oder weil ein Agent diesen Job ausführt und eine Statusmeldung darüber an den Server übermittelt. Um nicht jedes Mal die gesamte Jobtabelle durchsuchen zu müssen, wurde eine Indextabelle realisiert, die zu jeder Job-ID die Zeilennummer der Tabelle liefert, in der der Job gespeichert ist. Die Zeilen einer Tabelle kann man direkt anspringen, die Indextabelle wurde als Hashtabelle realisiert, so dass der Zeitaufwand zum Finden eines Jobs in der unsortierten Tabelle von $O(n)$ auf $O(1)$ sinkt. Die Indextabellen müssen jedoch immer dann neu aufgebaut werden, wenn sich die ID eines Jobs oder seine Position in der Tabelle ändert. Dies ist jedes Mal der Fall, wenn Jobs entfernt werden, neue Jobs dazukommen oder vorhandene Jobs eine neue ID erhalten. Der Neuaufbau der Indextabelle hat zwar lineare Komplexität, muss aber vergleichsweise selten durchgeführt werden. Das Ergebnis ist in Abb. 4.3 zu sehen. Ein einzelner Job wird bei den getesteten Gesamtjobmengen ohne messbaren Zeitbedarf gefunden, wohingegen ohne diesen Cache bei 10000 Jobs rund 0,1 und bei 1 Million Jobs ca. 1 Sekunde benötigt werden.

Eine weitere Hashtabelle speichert zu jedem Job die Information, ob er fertig bearbeitet wurde. So kann sehr schnell festgestellt werden, ob die Vorgängerbedingungen eines Jobs erfüllt sind. Da dieser Test bei der Erstellung der Kandidatenliste (vgl. Abschnitt 3.3.2) für den Scheduler pro Job einmal durchgeführt werden muss, kann durch diese Hilfsstruktur eine lineare Komplexität gehalten werden, ohne die Hashtabelle würde sie im ungünstigsten Fall quadratisch werden.

[10] Alle Tests wurden auf einem Server mit der Konfiguration AMD Athlon 2 GHz, 1 GB RAM, Windows Server 2003, .NET 2.0 durchgeführt

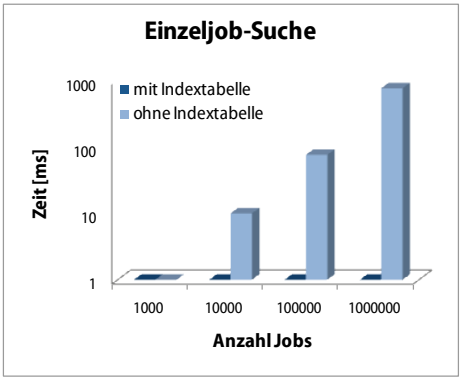


Abb. 4.3 Einfluss von Indextabellen

Interne Caches

Unter den Tests, die entscheiden, ob ein Job in die Kandidatenliste für den Scheduler aufgenommen wird oder nicht, sind auch zwei, die sehr zeitaufwändig sein können, da sie von der Festplatten- bzw. der Dateisystemgeschwindigkeit abhängen:

- Der Check, ob die bei einem Job angegebenen Quelldateien vorliegen, und
- die Ermittlung der Gesamtdatenmenge, die der Agent herunterladen müsste.

Stellt man diese Informationen zu jedem Job einzeln zusammen, wird das zum Problem, wenn man sehr viele Jobs prüfen muss. Das Dateisystem bzw. die Festplatte des Servers liefern diese Daten nicht schnell genug. Abhilfe schaffen wiederum als Hashtabellen implementierte Caches, die zu jeder einmal geprüften Quelldateiliste deren Vorhandensein und deren Gesamtgröße speichern. Da sich diese Listen und der Zustand auf dem Dateisystem selten ändern, sparen die beiden Caches enorm viele Festplattenzugriffe, was die Bildung der Kandidatenliste stark beschleunigt. Um trotzdem auf Veränderungen im Dateisystem, vor allem auf das Löschen von Dateien, reagieren zu können, werden die beiden Caches periodisch (beim Update der Jobcountdowns, vgl. Abschnitt 3.3.3) geleert. Bei den Messungen ergaben sich durchschnittliche Laufzeitverbesserungen um einen Faktor von rund 200 (Abb. 4.4): Um bei 10000 Jobs das Vorhandensein bzw. die Größe einer Quelldatei zu prüfen, werden nun nicht mehr 3 Sekunden, sondern nur noch rund 15 Millisekunden benötigt.

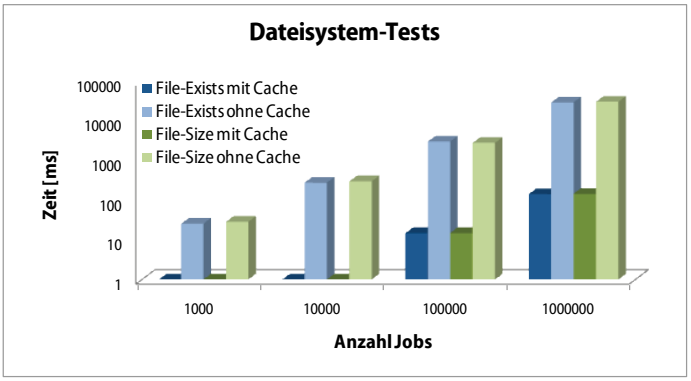


Abb. 4.4 Einfluss von Dateisystem-Caches

Hat der Scheduler dann einen Job ausgewählt, der an den anfragenden Agenten ausgeliefert wird, müssen noch die MD5-Prüfsummen aller Quelldateien des Jobs gebildet werden (siehe Abschnitt 4.3.3). Bei großen Dateien kostet dies viel Zeit, denn der Dateiinhalt muss komplett gelesen und seine MD5-Summe berechnet werden, was sowohl die Festplatte als auch die CPU des Servers belastet. Zwei weitere als Hashtabelle realisierte Cachespeicher, die zu jeder Datei deren letztes Änderungsdatum und den MD5-Wert speichern, lösen das Problem.

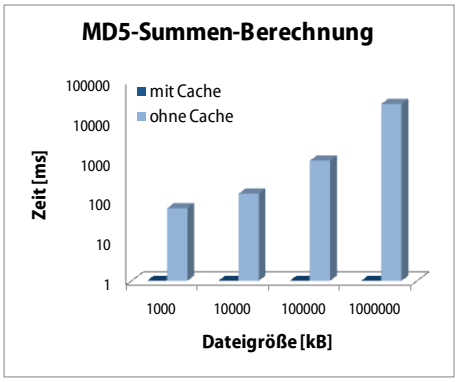


Abb. 4.5 Einfluss des MD5-Summen-Caches

Wie aus Abb. 4.5 ersichtlich wird, sinkt die wiederholte Berechnungszeit für eine 100 MB große Datei von ca. 1 Sekunde auf 0 ab. Handelt es sich um Dateigrößen, die nicht mehr in den Dateisystemcache des Betriebssystems passen, wird die relative Ersparnis noch größer. Die MD5-Summe einer 1 GB großen Datei würde ohne Cache erst nach ca. 25–30 Sekunden berechnet sein. In der Praxis wirkt sich dies jedoch nur dann aus, wenn man viele Jobs mit großen Datenmengen verteilen muss.

4.3.2 Messung von Bearbeitungszeiten und Anpassung der Timer-Parameter

Wie im Abschnitt 3.3.3 über Fehlertoleranz beschrieben wurde, läuft in der Serverkomponente ein Hintergrundthread, der in regelmäßigen Abständen ein Countdown-Update anstößt. Die Routine bearbeitet bei allen arbeitenden Jobs einige Datenfelder. Die für diese Routine benötigte Zeit wird gemessen. Dauert die Bearbeitung zu lange (über 2 bzw. über 5 Sekunden), wird das Intervall, in dem der Timer diese Routine auslöst, verdoppelt bzw. vervierfacht. Dadurch wird die Datenbasis weniger oft für den Zugriff der anderen serverinternen Routinen gesperrt.

Zwei weitere Maßnahmen betreffen die Agenten auf den Rechenknoten. Dort läuft während der Jobausführung ein Hintergrundthread, der regelmäßig den Server kontaktiert und diesen über den Fortschritt der Jobausführung informiert. Die für diesen Statusaustausch verwendete Zeit wird gemessen, entsprechend der Dauer wird das Intervall dann in Stufen (2–8 Sekunden) verlängert. Weiterhin wird die Zeit bestimmt, die benötigt wird, um sich vom Server einen Rechenjob zu holen. Übersteigt sie eine halbe Minute, so holt sich der Agent nach der erfolgreichen Ausführung eines Jobs nicht sofort einen neuen Auftrag, sondern wartet eine zufällige Zeitspanne (30–60 Sekunden), bevor eine neue Anfrage erfolgt. Beides bewirkt eine Verringerung der Kommunikationslast des Servers

und auch hier wird dessen Datenbasis weniger oft für den Zugriff der anderen serverinternen Routinen gesperrt.

4.3.3 Dateicaching

Eine typische Eigenschaft der einzelnen Rechenjobs eines Benutzers liegt darin, dass sich zwar alle Jobs in der Parametrisierung ihrer Algorithmen voneinander unterscheiden, die benötigten Quelldateien jedoch für alle Jobs häufig (zumindest in Teilen) identisch sind. Um für die Ausführung aufeinanderfolgender Jobs nicht wiederholt die gleichen Dateien vom Server herunterladen zu müssen, wurde ein Caching-Mechanismus implementiert, der genau dieses Problem angeht. Der Agent speichert alle heruntergeladenen Dateien in einem Cacheverzeichnis, welches (im Gegensatz zum Arbeitsverzeichnis des Jobs) nach der Jobausführung nicht gelöscht wird. Der Server schickt bei der Zuteilung eines Jobs an einen Agenten nicht nur die Namen der herunterzuladenden Dateien, sondern auch deren MD5-Hashwerte. Der Agent prüft nun, ob eine bereits vorhandene namensgleiche Datei die gleiche MD5-Summe besitzt und verwendet ggf. die lokal vorhandene Version. Der Caching-Mechanismus kostet zwar durch die Berechnung der MD5-Hashwerte ein wenig Rechenleistung, spart aber gerade bei schmalbandigen Netzzugängen je nach Dateigröße viel Übertragungszeit (Abb. 4.6), da anstelle des Downloads lediglich der Hashwert der lokal vorhandenen Datei berechnet wird und sie vom Cache ins Arbeitsverzeichnis kopiert werden muss.

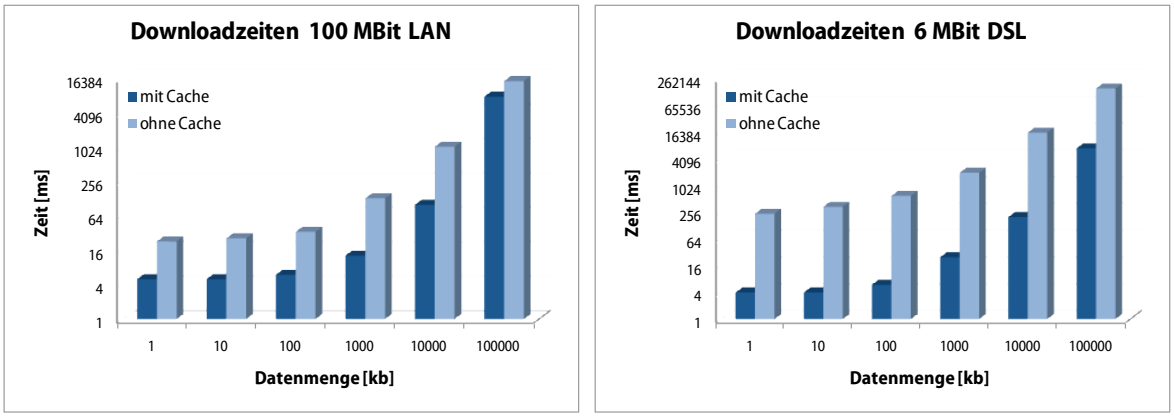


Abb. 4.6 Zeitbedarf von Dateidownloads

Die MD5-Werte werden serverintern nochmals in einem Cache verwaltet, so dass sie nicht bei jeder Agentenanfrage neu berechnet werden müssen (vgl. Abschnitt 4.3.1). Eine Neuberechnung der Prüfsummen führt der Server nur dann durch, wenn sich bei einer Datei der (ebenfalls in einer Hashtabelle protokollierte) Zeitstempel des letzten Schreibzugriffs geändert hat. In diesem Falle ist davon auszugehen, dass der Entwickler eine neue Version dieser Datei erstellt hat.

4.3.4 Dateikompression

Die Datenübertragung vom Server zum Agenten wird mittels des im vorigen Abschnitt beschriebenen Caching-Mechanismus beschleunigt. Bei der Übertragung von Ergebnis-

dateien vom Agenten zum Server kann ein derartiges Verfahren jedoch nicht eingesetzt werden, da davon ausgegangen werden muss, dass der Entwickler einer parallelen Anwendung jeder Ergebnisdatei einen eindeutigen Namen erteilt und sich bei aufeinanderfolgenden Jobs des gleichen Benutzers auch deren Inhalt ändert. Deswegen muss der Agent grundsätzlich alle Dateien zum Server laden. Um beim hier verwendeten HTTP-POST-Request nicht zu schnell an Timeoutgrenzen des Webservers zu stoßen, muss die Übertragungszeit einer einzelnen Datei so kurz wie nur möglich gehalten werden. Deswegen werden die zu übertragenen Ergebnisdateien vor dem Upload in ein zip-Archiv verpackt, welches der Webserver nach erfolgreichem Upload in das Zielverzeichnis des Entwicklers dekomprimiert (Abb. 4.7). Auch hier wird also ein wenig Rechenleistung zugunsten eines schnelleren Dateitransfers geopfert.

Die Dateien, die bei den Leistungstests verwendet wurden, waren intern so aufgebaut, dass sie sich auf ein Drittel ihrer ursprünglichen Größe komprimieren ließen, was die bei der langsamen DSL-Verbindung fast dreifache Uploadgeschwindigkeit erklärt. Die Abb. 4.7 macht aber auch deutlich, dass der Kompressions- und Dekompressionsaufwand nicht zu unterschätzen ist. Beim Einsatz im schnellen LAN kann die dort ebenfalls auf 30% gesunkene Übertragungszeit den durch die Kompression erhaltenen Zeitverlust nicht ausgleichen, der gesamte Upload dauert dort länger. Da die Übertragungszeiten im LAN aber insgesamt vergleichsweise kurz sind, wurde auch dort die Kompression für Dateien ab einer Größe von 1 Megabyte aktiviert.

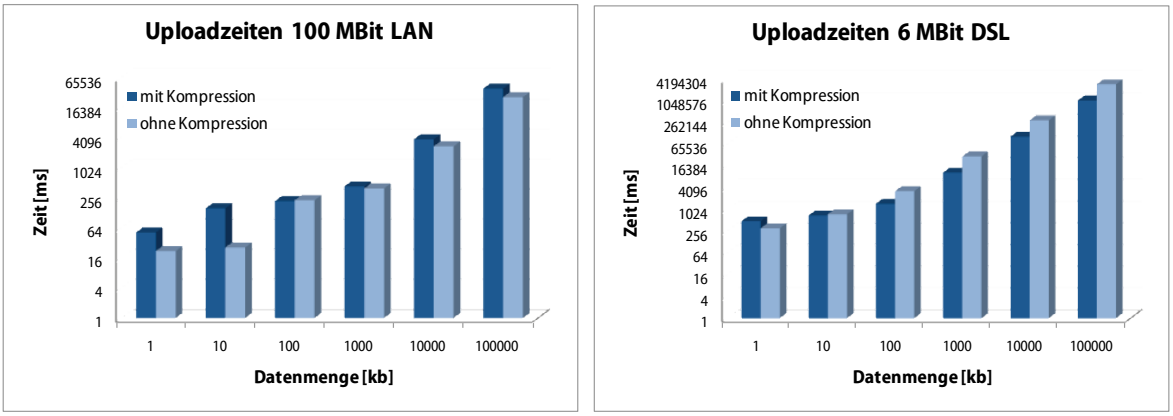


Abb. 4.7 Zeitbedarf von Dateiuploads

4.4 Betrieb eines JoSchKa-Servers

Wie in den Abschnitten 3.3.1 und 4.1 schon beschrieben, handelt es sich beim Server um eine ASP.NET-Webanwendung, sowie um Dateisystemfreigaben, die den Dateiaustausch per SMB ermöglichen. Dieses Kapitel beschreibt die Einrichtung und Konfiguration des Servers. Dabei wird davon ausgegangen, dass es sich um einen Windows Server 2003 handelt. Auf die Installation und die weitere Konfiguration des Betriebssystems soll dabei nicht näher eingegangen werden.

4.4.1 Nutzerkonten

Auf dem Server sind verschiedene Benutzerkonten anzulegen:

- Ein Konto mit normalen Benutzerrechten für jeden Entwickler, der das JoSchKa-System benutzen möchte. Der gewählte Benutzername ist vom Entwickler später auch bei der Definition von Jobs zu verwenden.
- Ein weiteres Konto mit normalen Benutzerrechten, dessen Benutzername *Q* sein muss. Dieses Konto erhält vom Server im Betrieb erweiterte Rechte und dient ausschließlich administrativen Zwecken. Beispielsweise kann man mit diesem Konto über das Management-GUI die Jobs aller Benutzer verwalten, nicht nur die eigenen.
- Ein zusätzliches Konto mit normalen Benutzerrechten, bei dem der Benutzername *nullnull7* lauten muss. Diesem Konto ist ein festes Passwort zu geben, welches man erhält, wenn man das Programm *DistProtoAgentGetPW.exe* ausführt. Dieses Passwort ist im Agenten fest einkompiliert und damit quasi-öffentlich. Das Konto wird von den Agenten benutzt, um sich beim Server zu authentifizieren, da keine Schnittstelle anonymen Zugriff erlaubt. Jobs auf dem Server zu hinterlegen oder vorhandene Jobs zu manipulieren, ist mit diesem Konto allerdings nicht möglich.

4.4.2 Webanwendung

Die Webapplikation stellt die SOAP-, die Download- und die Upload-Schnittstelle bereit, über die sich die Agenten mit Jobdaten versorgen und über die die Ergebnisse wieder an den Server übertragen werden. Bei der Installationsbeschreibung wird davon ausgegangen, dass auf dem Betriebssystem bereits der notwendige Webserver IIS (in Version 6) installiert ist. Ist dies nicht der Fall kann man das über *Systemsteuerung → Software → Windowskomponenten hinzufügen → Anwendungsserver* nachholen. Dabei ist zu beachten, dass die Managementkonsole und ASP.NET mit installiert werden. Sollte es nicht bereits vorhanden sein, ist auch das .NET-Framework in Version 2.0 zu installieren, da die Webanwendung mit der vom Betriebssystem mitgelieferten Version 1.1 nicht läuft.

Zunächst erstellt man im Basisverzeichnis des Webservers – normalerweise ist das *C:\Inetpub\wwwroot* – das Anwendungsverzeichnis *\DistProtoService*, in das man die Anwendungsdateien kopiert. Danach öffnet man die Management-Konsole (*Systemsteuerung → Verwaltung → Computerverwaltung*) und wählt den Zweig *Dienste und Anwendungen → Internetinformationsdienste* aus. Im Kontextmenü des neu angelegten Verzeichnisses unterhalb der Standardwebseite öffnet man die Eigenschaften (Abb. 4.8 links) und stellt einen Anwendungsnamen ein. Ab jetzt verhält sich die Webseite dynamisch, die ASP.NET-Seiten werden beim ersten Aufruf kompiliert und als nativer Code ausgeführt. Bei den Berechtigungen reicht Leseberechtigung und Scriptausführung aus. Nun sollte die Anwendung beim Test der URL *http://servername/DistProtoService/Service1.asmx* bereits funktionieren. Eine potentielle Fehlerquelle liegt darin, dass auf dem Server beide .NET-Frameworks installiert sind und die Anwendung mit dem falschen Framework ausgeführt wird. Die verwendete Framework-Version kann in den Eigenschaften des virtuellen Verzeichnisses im Reiter ASP.NET eingestellt werden. Sollten auf dem Server mehrere

Webanwendungen unter verschiedenen Framework-Versionen ausgeführt werden, müssen die 1.1er Anwendungen und die 2.0er Anwendungen in jeweils getrennten so genannten *Application-Pools* ausgeführt werden.

Damit die Webanwendung Dateien erzeugen und ändern kann, braucht der ASP.NET-Arbeitsprozess (*w3wp.exe*) Dateisystem-Schreibrechte im Webverzeichnis. Der Prozess läuft unter dem Konto *NETZWERKDIENST* bzw. *NETWORKSERVICE*, diesem Konto gibt man Vollzugriff im Verzeichnis (Abb. 4.8 rechts). Dem gesamten Anwendungsverzeichnis entzieht man dann den anonymen Zugriff (in der Managementkonsole bei den Eigenschaften der Anwendung im Bereich *Verzeichnissicherheit* einstellbar).

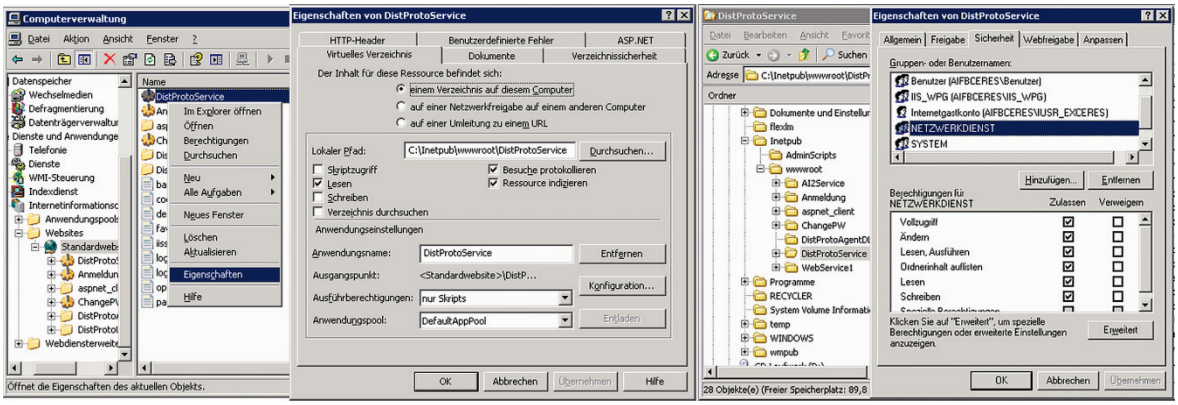


Abb. 4.8 Webanwendung erstellen und Dateisystemberechtigungen anpassen

Da der IIS 6 standardmäßig keine Dateien ausliefert, die keine Endung besitzen und auch keine Dateien, deren Datentyp er nicht kennt, muss eine entsprechende Konfigurationsänderung vorgenommen werden, sofern man beliebige Dateien zum Download anbieten will. Dazu öffnet man wieder die Internetinformationsdienste-Verwaltung, wählt die Eigenschaften des virtuellen Verzeichnisses aus und erstellt im Bereich *HTTP-Header* den MIME-Type „.* application/octet-stream“ (Abb. 4.9).

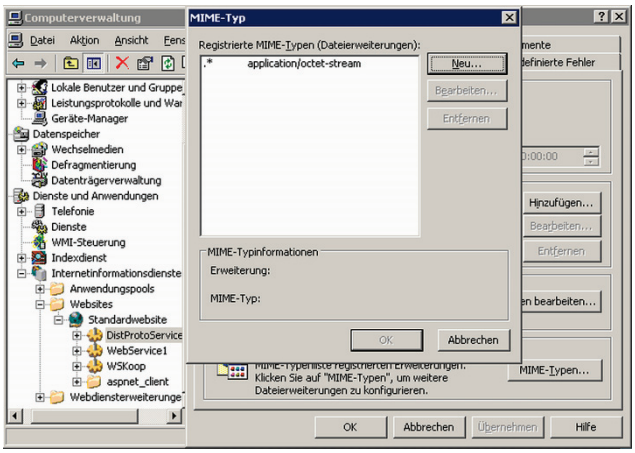


Abb. 4.9 MIME-Type einrichten

Zuletzt ist noch die maximale Länge von HTTP-Requests und deren Timeoutgrenze anzupassen. Da die Agenten später (unter Umständen sehr große) Dateien per HTTP-POST

an den Server übertragen, muss die maximale Länge, die der Webserver für solche Requests zulässt, und deren maximale Dauer erhöht werden. Dazu öffnet man mit einem Texteditor die anwendungsspezifische ASP.NET Konfigurationsdatei *web.config*, die sich im Hauptverzeichnis der Anwendung befindet, und ändert in der Sektion <*system.web*> (Listing 4.3) die maximale Anfragelänge und Zeitdauer ab (Werte in Kilobytes bzw. Sekunden).

Listing 4.3 ASP.NET-Konfiguration

```
<system.web>
     ...
     <httpRuntime ... executionTimeout="600" maxRequestLength="262144" ... />
     ...
</system.web>
```

Damit akzeptiert der Server Ergebnisdateien mit einer Größe von bis zu 256 Megabytes, der Upload darf dabei 10 Minuten dauern. Sollte das nicht ausreichen, sind die Werte entsprechend anzupassen. Die Änderungen können auch an zentraler Stelle erledigt werden und sind dann für alle Anwendungen gültig: Vor Änderungen an den globalen Konfigurationsdateien *machine.config* und *web.config*, die normalerweise im Verzeichnis *C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG* liegen, sollte man jedoch auf jeden Fall ein Backup dieser Dateien anlegen[11].

Die Webanwendung ist damit einsatzbereit, jetzt sollten die folgenden URLs funktionieren:

- *http://server/DistProtoService/Service1.asm*x
  SOAP-Schnittstelle, zeigt beim Aufruf mit dem Webbrowser die Webmethoden an
- *http://server/DistProtoService/FileUpload.aspx*
  Dateiupload, sollte beim Aufruf mit dem Webbrowser „BAD_QUERYSTRING“ liefern
- *http://server/DistprotoService/UploadAPIWorkerDLL.aspx*
  Uploadschnittstelle für die API-Worker-DLL, sollte beim Aufruf mit dem Webbrowser „BAD_QUERYSTRING“ liefern
- *http://server/DistProtoService/JobStatus.aspx*
  Webformular zur Ansicht der Jobdatenbasis per Webbrowser
- *http://server/DistProtoService/AgentStatus.aspx*
  Webformular zur Ansicht der Agenten per Webbrowser
- *http://server/DistProtoService/NodeStatus.aspx*
  Webformular zur Ansicht der Rechenknoten per Webbrowser
- *http://server/DistProtoService/GetFullDatabaseDump.aspx*
  Vollständige Jobdaten im XML-Format, nur administrativem Benutzer zugänglich

[11] Umfangreiche Informationen zur Konfiguration und dem Deployment von ASP.NET Webanwendungen findet man z. B. in [Sch06b]

Anschließend müssen noch die Schnittstellen für die Ein- und Ausgabedateien eingerichtet, sowie einige weitere Konfigurationsoptionen geprüft und gegebenenfalls angepasst werden. Dies wird in den nächsten beiden Abschnitten erläutert.

4.4.3 Schnittstellen für Ein- und Ausgabedateien

Die Ein- und Ausgabedateien werden über eine Netzwerkfreigabe auf den Server übertragen bzw. von dort abgeholt. Hierfür sind pro Benutzer zwei Freigaben einzurichten. Die Freigabe für die Eingabedateien befindet sich im Verzeichnis der Webanwendung, da die Agenten per HTTP diese Dateien ja ebenfalls erreichen müssen. Im Verzeichnis *\FileDownload* der Webanwendung legt man für jeden Benutzer ein Verzeichnis an, das den gleichen Namen hat wie die Benutzer-ID, also z. B. *\FileDownload\alba*. Der zugehörige Benutzer benötigt Vollzugriff auf dieses Verzeichnis. Die Dateisystem- und Freigabenrechte sind entsprechend anzupassen (Abb. 4.10). Die Freigabe ist dann von einem Windowsrechner aus mit dem Befehl bzw. unter der Adresse *\\server\alba* erreichbar. Unter Linux erreicht man eine solche Freigabe mit dem Tool *smbclient*.

Bei der SMB-Freigabe für das Zielverzeichnis ist entsprechend gleich vorzugehen, lediglich der Freigabename ist anders zu wählen, etwa „alba_results“. Dieses Verzeichnis muss sich nicht notwendigerweise im Pfad der Webanwendung befinden. Es ist aber darauf zu achten, dass das Benutzerkonto *NETZWERKDIENST* auf diesem Verzeichnis ebenfalls Schreibrechte benötigt, da dort die Ergebnisdateien der Jobs abgespeichert werden. Zugriff auf die Ergebnisdateien erhält man analog unter *\\server\alba_results*.

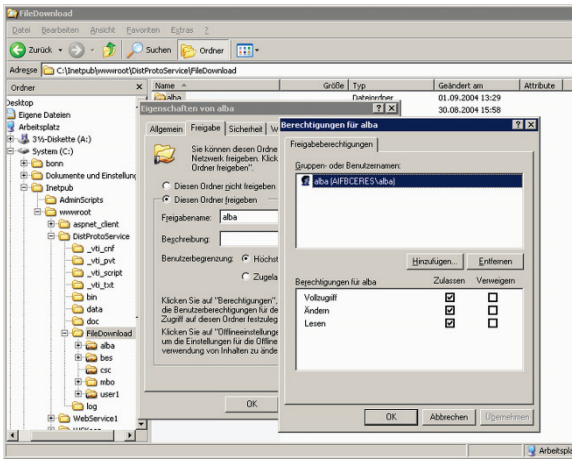


Abb. 4.10 Freigabe für Quelldateien

4.4.4 Weitere Konfiguration

Im Basisverzeichnis der Webanwendung befindet sich die Datei *appConfig.xml*, mit der einige Parameter des Servers gesteuert werden. Listing 4.4 zeigt zunächst eine beispielhafte Konfiguration mit typischen bzw. sinnvollen Voreinstellungen, anschließend werden die einzelnen Parameter erläutert.

Listing 4.4 Konfiguration der Webanwendung

```
<?xml version="1.0" encoding="utf-8"?>
<appConfig>
     <timerIntervall>240000</timerIntervall>
     <reloadMultiplikator>5</reloadMultiplikator>
     <resultsPath>D:\DistProtoResults\</resultsPath>
     <notificationSenderAdress>bonn@aifb.uka.de</notificationSenderAdress>
     <notificationSmtpServer>smtp.uni-karlsruhe.de</notificationSmtpServer>
     <fairLevel>10</fairLevel>
     <doneRateLowBoost>3</doneRateLowBoost>
     <powerIndexProb>0</powerIndexProb>
     <runlengthScale>0</runlengthScale>
     <useUptimes>YES</useUptimes>
     <redundantDelivery>YES</redundantDelivery>
     <persistentGET>NO</persistentGET>
     <persistentCOM>YES</persistentCOM>
     <persistentEXC>NO</persistentEXC>
     <verboseLog>NO</verboseLog>
</appConfig>
```

Die Tags bedeuten im Einzelnen:

- *timerIntervall:* Angabe in ganzzahligen Millisekunden, die der Server zwischen zwei Countdown-Updates wartet. Zu den Countdown-Updates siehe auch Abschnitt 3.3.3.
- *reloadMultiplikator:* Gibt an, bei jedem wievielten Countdown-Update die Datendatei neu eingelesen werden soll. Erlaubt sind ganzzahlige positive Werte.
- *resultsPath:* Ort im Dateisystem, an dem die Ergebnisdateien abgespeichert werden (vgl. vorherigen Abschnitt).
- *notificationSenderAdress:* Mit dieser Absenderadresse werden die Benachrichtigungsemails verschickt.
- *notificationSmtpServer:* Über diesen Mailserver werden die Benachrichtigungsmails verschickt.
- *fairLevel*, *doneRateLowBoost*, *powerIndexProb* und *runlengthScale:* Dabei handelt es sich um die Schedulingparameter, die den kombinierten Verteilalgorithmus steuern. Zu diesen Parametern siehe Abschnitt 3.3.5f. Die Parameter werden in der Konfigurationsdatei allerdings nicht als reellwertige Wahrscheinlichkeiten, sondern als ganzzahlige Prozentwerte angegeben, wobei das Prozentzeichen weggelassen wird. Soll beim laufzeit- und beim betriebszeitbasierten Verteilverfahren die Spreizung dynamisch erfolgen, ist *runlengthScale* auf den Wert −1 zu setzen.
- *useUptimes*: Dieser Parameter legt fest, ob beim Verteilalgorithmus das betriebszeitbasierte Verfahren anstelle des laufzeitbasierten eingesetzt werden soll. Gültige Werte sind YES und NO.
- *redundantDelivery*: Mit diesem Schalter wird die Mehrfachauslieferung vom Jobs (vgl. Abschnitt 3.3.8) (de-)aktiviert. Gültige Werte sind YES und NO.

- *persistentGET*, *persistentCOM* und *persistentEXC:* Die Parameter legen fest, ob nach Ausführung der SOAP-Funktionen *GetJob()*, *CommitJob()* und *ExchangeStatus()* die Datenbasis persistent auf die Festplatte serialisiert werden soll. Aus Performancegründen ist dies im Beispiel nur bei der wichtigsten Funktion, dem Commit, aktiviert. Unabhängig davon schreibt der Server bei jedem Countdown-Update und beim Shutdown die Daten mit Backupnummer auf die Platte. Gültige Werte für die Parameter sind ebenfalls YES und NO.
- *verboseLog:* Der Parameter steuert den Detaillierungsgrad der Logfiles. Gültige Werte für die Parameter sind wiederum YES und NO.

4.4.5 Webverzeichnis für automatisches Programmupdate

JoSchKa besitzt die Fähigkeit, alle Client-Programme automatisch beim Start auf die neueste, auf dem Server bereitliegende Version des jeweiligen Programms zu aktualisieren. Automatisch aktualisiert werden können neben dem Managertool für den Nutzer (vgl. Kapitel 3.6.2) auch die Agenten, die die Jobs ausführen. Um die Updatefunktion zu nutzen, ist im Wurzelverzeichnis des Webservers ein neues Verzeichnis anzulegen (z. B. *C:\Inetpub\wwwroot\DistProtoUpdate*), in das man alle Programmdateien ablegt. Für jede Anwendung ist eine XML-Datei mit Namen *<programmname>.updateconf.xml* anzulegen, die im Falle von z. B. *DistProtoAgent.updateconf.xml* den in Listing 4.5 gezeigten Inhalt hat.

Listing 4.5 Konfiguration für automatisches Agenten-Update

```
<?xml version="1.0" encoding="utf-8"?>
<config>
 <dl>
  <file name="DistProtoAgent.exe"
   url="http://server/DistProtoUpdate/DistProtoAgent.exe"/>
  <file name="DistProtoAgent.XmlSerializers.dll"
   url="http://server/DistProtoUpdate/DistProtoAgent.XmlSerializers.dll"/>
  <file name="ICSharpCode.SharpZipLib.dll"
   url="http://server/DistProtoUpdate/ICSharpCode.SharpZipLib.dll"/>
 </dl>
 <startCommand>DistProtoAgent.exe</startCommand>
</config>
```

Diese XML-Datei wird vom Updateprogramm heruntergeladen, aus ihr entnimmt es die Informationen, aus welchen Teilkomponenten der Rechenagent besteht und wie er zu starten ist. Für die anderen JoSchKa-Programme *DistProtoScreensaver* und *DistProtoManager* existieren ebenfalls entsprechende Dateien.

4.5 Betrieb eines JoSchKa-Agenten

Das Agentenprogramm wurde ohne graphische Benutzeroberfläche als reines Kommandozeilentool (*DistProtoAgent.exe*) realisiert, was ein automatisches Starten des Programms beim Systemstart, also quasi als Hintergrunddienst, ermöglicht. Es wird deswegen

ausschließlich per Konfigurationsdatei und/oder Kommandozeilenparameter gesteuert und besitzt keinerlei interaktive Komponenten.

Im Weiteren wird zunächst beschrieben, wie ein JoSchKa-Agent konfiguriert wird und welche weitere Möglichkeiten die ebenfalls realisierte graphische Agentenversion *DistProtoScreensaver.exe* bietet. Anschließend folgt eine Beschreibung, wie mehrere Agenten parallel auf einem Server als Hintergrunddienst eingerichtet werden, wobei die Agenten automatisch mit der jeweils neuesten Version gestartet werden.

Grundsätzlich bestehen die Agenten aus den folgenden Dateien:

- *DistProtoAgent.exe*/*DistProtoAgent.XmlSerializers.dll*: Das kommandozeilenorientierte Agentenprogramm und seine Webserviceclient-Bibliothek
- *DistProtoScreensaver.exe*/*DistProtoScreensaver.XmlSerializers.dll*: Das graphische Agentenprogramm und seine Webserviceclient-Bibliothek
- *DistProtoAgent.config.xml*/*DistProtoScreensaver.config.xml*: Konfigurationsdateien für die Agentenprogramme
- *ICSharpCode.SharpZipLib.dll*: Fremdbibliothek für den komprimierten Dateiupload
- *pssuspend.exe*/*Win32_API.dll*: Fremdprogramme/-bibliotheken für den graphischen Agenten zum Einfrieren von Jobs und zum Messen der Idle-Zeit

4.5.1 Konfiguration und offener Betrieb

Die Konfigurationsdatei *DistProtoAgent.config.xml* wird beim Start eingelesen und sieht typischerweise wie in Listing 4.6 angegeben aus.

Listing 4.6 Agentenkonfiguration

```
<?xml version="1.0" encoding="utf-8"?>
<config>
     <loopCount>-1</loopCount>
     <jobType></jobType>
     <agentPlat></agentPlat>
     <url>http://ceres.aifb.uka.de/DistprotoService</url>
     <multiMode>NO</multiMode>
     <cacheFiles>YES</cacheFiles>
     <startProcessHidden>NO</startProcessHidden>
     <agentdirectory>C:\temp\</agentdirectory>
     <activeHours>00,01,02,03,04,05,06,07,18,19,20,21,22,23</activeHours>
     <maxDLSize></maxDLSize>
</config>
```

Die Tags bedeuten im Einzelnen:

- *loopCount:* Anzahl der Jobs, die der Agent nacheinander auszuführen versucht. Soll der Agent in Endlosschleife laufen, ist eine −1 einzutragen.

- *jobType:* Ermöglicht es, den Agenten auf einen bestimmten Benutzer bzw. Typ festzulegen. Soll der Agent universell agieren, ist das Feld leer zu lassen. Möchte man Prioritäten vergeben, gibt man die entsprechenden Job-Typen semikolongetrennt hintereinander an, z. B. „mbo_diss;aka_projekt;“ würde dafür sorgen, dass der Agent zunächst beim Server einen Job des Typs „mbo_diss“ erfragt. Sollten beim Server keine Jobs dieses Typs verfügbar sein, verlangt der Agent einen des Typs „aka_projekt“. Im Falle eines weiteren Fehlschlages würde der Agent eine Anfrage nach einem beliebigen Job stellen.
- *agentPlat:* Ermöglicht es, den Agenten auf eine bestimmte Plattform zu zwingen bzw. Jobs für bestimmte Plattformen nicht auszuführen. Dabei stehen W für Windows (nativ), L für Linux (nativ), N für .NET (auf Windows), M für Mono (auf Linux), J für Java, P für Perl, Y für Python und R für R. Für Kombinationen sind die einzelnen Zeichen einfach aneinanderzuhängen, z. B. „WNJ“ für einen Agenten, der auf einer Windowsmaschine läuft, die auch .NET und Java-Programme ausführen kann. An diese Angabe können noch (durch jeweils einen Punkt abgetrennt) Angaben über den Hauptspeicher, den der Job nicht überschreiten soll, und die maximale Uploaddatenmenge sowie die CPU-Architektur angehängt werden, z. B. „WNJ.256.10.x86“ für einen Agenten, der nur Jobs ausführen soll, die nicht mehr als 256 MB Hauptspeicher benötigen, maximal 10 MB Daten erzeugen und auf einem 32-Bit System laufen. Bleibt das Feld leer, bestimmt der Agent selbst, welche Laufzeitumgebungen vorhanden sind und über wie viel RAM der Rechner verfügt (vgl. Abschnitt 3.4.1). Dies ist auch die empfohlene Einstellung.
- *url:* Die Serveradresse. Da der Agent Teile der URL zur Laufzeit dynamisch erstellen muss (vgl. Abschnitte 3.4.2 und 3.4.4), ist diese nur bis zur Webanwendung anzugeben, keinesfalls die komplette SOAP-URL (diese würde für obiges Beispiel *http://ceres.aifb.uka.de/DistprotoService/Service1.asmx?WSDL* heißen).
- *mulitMode:* Falls YES eingestellt ist, kann der Agent mehrfach auf dem gleichen Rechner unter Verwendung des gleichen Arbeitsverzeichnisses/Caches gestartet werden, die Default-Einstellung ist NO.
- *cacheFiles:* Steuert das Cachingverhalten des Agenten. Falls NO gewählt ist, wird der lokale Dateicache nicht verwendet und alle Dateien werden jedes Mal neu vom Server heruntergeladen. Die Default-Einstellung ist YES, Caching ist also aktiv.
- *startProcessHidden:* Bedeutet, dass der eigentliche Rechenprozess im Hintergrund gestartet wird. Das spielt nur dann eine Rolle, wenn der Agent interaktiv in einer Konsole gestartet wird. Läuft der Agent selbst im Hintergrund, werden die Rechenjobs grundsätzlich verdeckt gestartet.
- *agentDirectory:* Das Arbeitsverzeichnis des Agenten. Hier (und nur hier) braucht das Programm Schreibrechte. Das Arbeitsverzeichnis kann (und sollte) sich dabei vom Installationsverzeichnis unterscheiden.
- *activeHours:* Damit sind die Stunden des Tages gemeint, in denen der Agent arbeitet, also beim Server Jobs anfragt und diese ausführt. Im obigen Beispiel würde

der Agent von 08:00 bis 17:59 keine Jobs ausführen (evtl. schon laufende Jobs werden allerdings ordentlich beendet).

- *maxDLSize:* Die maximale Datenmenge in KB, die der Agent herunterladen darf. Trägt man hier ein −1 ein, existiert keine Größenbeschränkung. Lässt man das Feld leer, misst der Agent selbst die Netzwerkbandbreite aus. Abhängig davon, wie lange der Download zweier 500 KB-Dateien vom Server dauert (10 s, 1 min, 5 min) wird die maximale Datenmenge auf 20 MB, 5 MB oder 1 MB festgelegt.

Diese Einstellungen können mit Kommandozeilenoptionen überschrieben werden. Der Befehl *DistProtoAgent.exe /?* liefert die Syntax. Die einzelnen Optionen orientieren sich an den Möglichkeiten der Konfigurationsdatei und sind selbsterklärend.

Wird der Agent dann in einer offenen Konsole gestartet (Listing 4.7), zeigt er typischerweise die folgende Ausgabe (die Kernaufgaben des Agenten sind hervorgehoben):

Listing 4.7 Betrieb eines Agenten in einer Konsole

```
Bonn@AIFBACHERON <D:\DistProtoRelease>DistProtoAgent.exe
Testing Environment...
OS: Microsoft Windows NT 5.1.2600 Service Pack 2
OS Platform: Win32NT
OS Version: 5.1.2600.131072
CLR Version: 2.0.50727.42
Assembly-Info: DistProtoAgent, Version=2.5.2438.23222, Culture=neutral, Pub-
licKeyToken=null
Total physical Memory: 768 MB
Testing for Java...True
Testing for Perl...False
Testing for Python...False
Testing for R...False
Reading config.xml...OK
Initializing Communication...WebClient OK, URL OK, PreAuthenticate OK
--------ConfigState--------
OS: WINNT
jobType:
agentPlat: WNJ.768.*.x86
loopCount: -1
url: http://bceres.aifb.uka.de/DistprotoService
mode: s
hidden: False
cacheFiles: True
bossflag: False
dnsname: AIFBACHERON
machine: AIFBACHERON\Bonn.2212
actualLogFile: AgentLog_2006-10-4_18-20-53.log
baseDirectory: C:\temp\DPA\
cacheDirectory: C:\temp\DPA\cache\
workingDirectory: C:\temp\DPA\workdir_AIFBACHERON\
activeHours: 00,01,02,03,04,05,06,07,18,19,20,21,22,23
-------------------------
=> TESTING SYSTEM PERFORMANCE...
Floatingpoint-Array... 2383
Integer-Matrix... 8221
Matrix-Search... 2673
N-Dame Recursion... 12197
```

```
=> BENCHMARK AVG RESULT: 6368
SENDING BENCHMARK RESULT TO SERVER...
created new node with bench 6368
Testing Download-Rate... Time: 20
Resulting maxDLSize: -1
=> CREATED EMPTY WORKING DIRECTORY
Checking Time: 18 in 00,01,02,03,04,05,06,07,18,19,20,21,22,23 True
================ NEW JOB ================
=> LOCKED CACHE
=> REQUESTING FOR JOB, GOT THIS FROM SERVER:
JOB_OK
mbo_bench
09091646-81200947950400008
bench.exe a 3 6300
*
bench.exe
NO
3438
Bench1
B75F53000E4B3FF10B987853DD4EDCF3
=> DOWNLOADING FILES:
http://ceres.aifb.uka.de/DistProtoService/FileDownload/mbo/bench/bench.exe
...is cached.
Copied bench.exe
=> UNLOCKED CACHE
=> TOTAL TIME FOR GETTING JOB: 70 ms
=> STARTING JOB:
bench.exe a 3 6300
=> JOB STATUS: runs 00000 min
=> STATUS EXCHANGE WITH SERVER: OK (Time: 20 ms)
=> LAST MINUTE AVAILABLE FREE MEMORY MB: 355
=> LAST MINUTE AVERAGE CPU THREAD QUEUE: 5
=> CPU QUEUE LOW
=> JOB ENDED WITH EXIT-CODE 0, USED TIME:
00:02:39.3591472
=> SENDING OUT, ERROR AND EXIT:
FILE_SAVED C:\DistProtoResults\mbo\bench\Bench1.ALL
=> SENDING RESULTS:
subdir\subsubdir\results.log:
FILE_SAVED C:\DistProtoResults\mbo\bench\subdir\subsubdir\results.log
subdir\results.log:
FILE_SAVED C:\DistProtoResults\mbo\bench\subdir\results.log
results.log:
FILE_SAVED C:\DistProtoResults\mbo\bench\results.log
=> TIME FOR FILE UPLOAD: 40 ms
=> TRYING TO COMMIT UPLOAD:
JOB_COMMITTED;mbo_bench;09091646-81200947950400008
=> CREATED EMPTY WORKING DIRECTORY
=> GOT THESE ORDERS: TYPE:
                     PLAT: WNJ.768.*.x86
Bonn@AIFBACHERON <D:\DistProtoRelease>
```

Während der Ausführung schreibt der Agent ein Logfile, das in seinem Arbeitsverzeichnis unter dem Namen *AgentLog_<Datum>-<Uhrzeit>.log* einsehbar ist.

Das Agentenprogramm wurde neben der Kommandozeilenversion zusätzlich in einer Variante mit graphischem Frontend (*DistProtoScreensaver.exe*, Abb. 4.11) realisiert, die

sich über die Datei *screensaverconfig.xml* konfigurieren lässt. Die Parameter und deren Bedeutung entsprechen denen der Kommandozeilenversion. Benennt man die Datei in *DistProtoScreensaver.scr* um, lässt sie sich auch als Bildschirmschoner betreiben.

Die Funktionalität, dass ein Job nur dann CPU-Last erzeugt, wenn der am Rechner interaktiv arbeitende Nutzer keine Tastatur- oder Mauseingaben tätigt, lässt sich jedoch auch ohne Bildschirmschoner-Modus erreichen. Im GUI hat man die Möglichkeit festzulegen, nach welcher Idle-Zeit der Job ausgeführt wird (*Idle Seconds*). Bewegt man dann wieder die Maus oder benutzt die Tastatur, wird der Job eingefroren. Damit ist sichergestellt, dass der Rechner wirklich nur dann für fremde Jobs genutzt wird, wenn der eigentliche Benutzer gerade nichts am Rechner tut. Dies wurde bis jetzt nur für die Windowsplattform realisiert. Als weiteres Zusatzfeature bietet die GUI-Variante die Möglichkeit, sich nach Beendigung des aktuell aktiven Jobs selbst zu beenden.

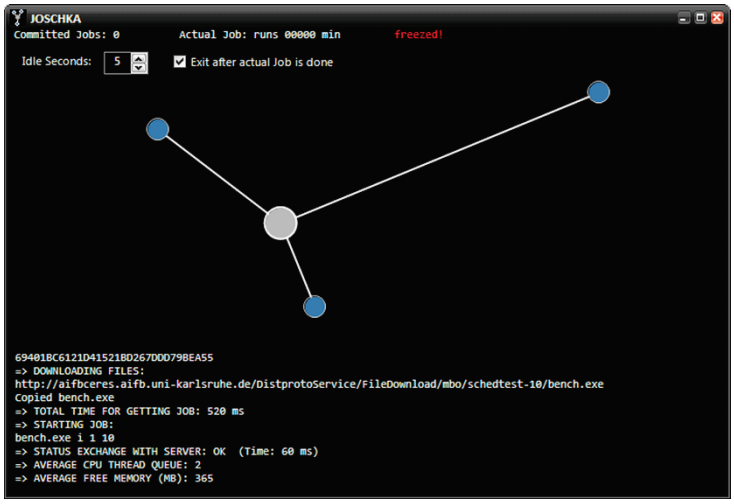


Abb. 4.11 Graphische Agentenversion

4.5.2 Hintergrundbetrieb mit automatischem Update

Interessant wird der Betrieb eines Agenten als Hintergrunddienst, wobei sein Start beim Systemstart automatisch erfolgen soll. Dabei wird vor dem Start die jeweils neueste Agentenversion vom Server heruntergeladen. Hierzu wird das Autoupdate- und Startprogramm *JoSchKaStarter.exe* eingesetzt. Das Agentenprogramm wird dabei nach dem Download mit eingeschränkten Benutzerrechten ausgeführt.

Im Folgenden Beispiel wird davon ausgegangen, dass es sich bei dem Rechner um einen Windows 2003 Server handelt, auf dem man zunächst Administratorrechte hat (im späteren Betrieb läuft der Agent natürlich mit eingeschränkten Rechten). Auf dem Server sei das .NET Framework 2.0 installiert. Der Agent soll in vier Instanzen gestartet werden, um die beiden installierten Doppelkernprozessoren durch vier gleichzeitig ausgeführte Jobs optimal auslasten zu können. Zwei der Agenteninstanzen sollen für alle Benutzer, zwei weitere jedoch ausschließlich für den Nutzer „mbo“ arbeiten.

Zunächst werden die in Listing 4.8 angegebenen Dateien auf dem Rechenknoten angelegt:

Listing 4.8 Dateien für Agentenbetrieb

```
C:\Programme\JoSchKa\JoSchKaStarter.exe
C:\Programme\JoSchKa\JoSchKaStarter.config.xml
C:\Programme\JoSchKa\DistProtoAgent.config.xml
```

DistProtoAgent.config.xml steuert das eigentliche Agentenprogramm und wird entsprechend dem vorangegangenen Abschnitt eingerichtet, *JoSchKaStarter.config.xml* legt die Update- und Startparameter fest und hat typischerweise folgenden den in Listing 4.9 angegebenen Inhalt.

Listing 4.9 Konfiguration für Agentenstart mit automatischem Update

```
<?xml version="1.0" encoding="utf-8"?>
<config>
     <baseURL>http://server/DistProtoUpdate/</baseURL>
     <localTargetDir>C:\temp\</localTargetDir>
     <startprofile match="4agents" args="-m" instances="2" />
     <startprofile match="4agents" args="-m -t=mbo" instances="2" />
</config>
```

Die Tags <*baseURL*> und <*localTargetDir*> legen fest, wie das Updateprogramm den Updateserver erreicht und in welches lokale Verzeichnis die Dateien abgelegt werden sollen (in diesem Verzeichnis braucht man Schreibrechte). Die <*startprofile*>-Einträge legen fest, mit welchen Parametern und wie oft das Agentenprogramm mit eben diesen Parametern jeweils gestartet werden soll. Ein Startprofil wird ausgewählt, in dem man den Wert des *match*-Attributs als Kommandozeilenparameter zusätzlich zum Namen des zu startenden Programms an *JoSchKaStarter.exe* übergibt. Dies kann über eine Kommandozeile oder eine entsprechende Verknüpfung geschehen:

```
JoSchKaStarter.exe DistProtoAgent 4agents
```

JoSchKaStarter.exe würde dann vom Updateserver (vgl. 4.4.5) die Datei

```
http://server/DistProtoUpdate/DistProtoAgent.updateconf.xml
```

und die darin spezifizierten Dateien herunterladen und in *C:\temp\DistProtoAgent* ablegen. Anschließend würde zweimal die Befehlszeile

```
DistProtoAgent.exe -m
```

sowie zweimal der Befehl

```
DistProtoAgent.exe -m -t=mbo
```

ausgeführt werden. Abschließend erstellt man in der Systemsteuerung einen neuen geplanten Task, der die folgende Befehlszeile startet:

```
C:\Programme\JoSchKa\JoSchKaStarter.exe DistProtoAgent 4agents
```

Der Task muss so eingestellt werden, dass er mit eingeschränkten Benutzerrechten läuft, beim Systemstart gestartet wird, nicht nach einer bestimmten Zeit abgebrochen wird und nach der Ausführung nicht entfernt wird (Abb. 4.12). Der Benutzer, unter dessen Account die Agenten laufen, sollte noch in die Benutzergruppe der Systemmonitorbenutzer aufgenommen werden, damit das Auslesen der Systemlast bei der Jobausführung funktioniert.

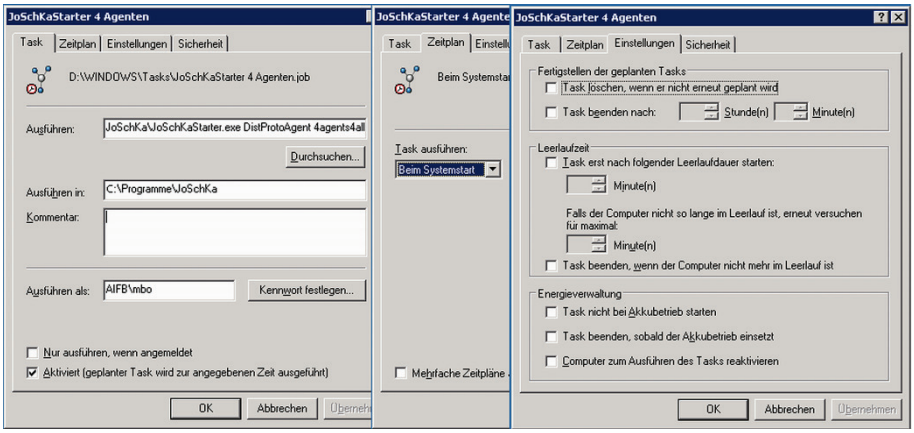


Abb. 4.12 Taskplan zum automatischen Agentenstart mit Update

Der beschriebene Autoupdatemechanismus kann auch verwendet werden, um das Management-GUI oder die graphische Agentenversion zu starten und dabei vorher zu aktualisieren.

4.6 Beeinflussung von Arbeitsplatzrechnern

Um den Einfluss des Agentenprogramms und der von ihm ausgeführten Jobs auf Arbeitsplatz oder Poolrechnern quantifizieren zu können [VySu07], wurde ein Benchmarkskript entwickelt, das nacheinander 10 Tests durchführt und deren jeweilige Ausführungsdauer misst. Bei den Tests wurden neben reinen arithmetischen Benchmarks auch alltägliche Aufgaben durchgeführt, etwa das Kopieren von Dateien auf Netzlaufwerke oder das Abspielen graphischer Anwendungen, wie beispielsweise eine Powerpoint-Präsentation. Die Tests wurden auf einem 1,5 GHz Pentium-M-System mit 768 MB Hauptspeicher unter Windows XP ausgeführt. Der Testrechner befand sich dabei in 3 unterschiedlichen Zuständen:

(a) ohne Hintergrundlast

(b) mit Hintergrundlast durch einen in Java geschriebenen JoSchKa-Job

(c) mit Hintergrundlast durch einen in C++ geschriebenen JoSchKa-Job

Die Jobs waren keine synthetischen Jobs, sondern reale Anwendungen, wie sie täglich über das System auf dem Pool (siehe Abschnitt 5.1) ausgeführt werden. Die ausgeführten Vordergrundanwendungen waren die folgenden:

- *CPU-Bench*: Dabei handelt es sich um einen arithmetisch-logischen Benchmark, wie er ebenfalls von den Agenten durchgeführt wird, wenn sie nach dem Start die Leistungsfähigkeit des Rechenknotens ausmessen (vgl. Abschnitt 3.3.4).

- *Web-Download*: Dieser Test führt HTTP-Downloads im 100 MBit-Netz durch: Es werden 3 kleine (~ 100–200 kB) und eine große (~ 40 MB) Datei heruntergeladen.
- *SMB-Copy*: Hierbei werden rund 180 MB Daten, verteilt auf Dateien von 1 kB bis 60 MB Größe, über ein 100 MBit-LAN auf eine SMB-Dateifreigabe kopiert.
- *Browser/GUI*: Zum Einsatz kommt eine graphische Anwendung, die Webseiten darstellt, dabei mehrere komplexe Benutzerschnittstellenfenster öffnet und sich danach selbst beendet.
- *Powerpoint*: Die Präsentationssoftware Powerpoint spielt eine Folienfolge ab, die verschiedene, komplexe Animationen enthält.
- *Acrobat-Start-Kill*: Das Programm Adobe Acrobat wird mehrfach gestartet und der jeweilige Systemprozess sofort wieder beendet.
- *Powershell-Skript*: Ein Powershell-Skript führt eine verschachtelte Schleifenoperation aus und schreibt die Ausgabe in eine Datei.
- *Powershell-Konsole*: Ein Powershell-Skript führt eine verschachtelte Schleifenoperation aus und gibt die Ausgabe auf der Konsole aus.
- *MD5-Hash*: Die MD5-Prüfsumme einer 40 MB großen Datei wird berechnet.
- *XML2HTML*: Eine rund 4 MB große XML-Datei wird mit Hilfe eines DOM-Parsers gelesen, gefiltert, sortiert und als HTML-Datei wieder ausgegeben.

Die Tests beanspruchen also völlig unterschiedliche Bereiche des Systems. Neben der reinen CPU-Leistung (z. B. für Matrixmultiplikation oder kryptographische Aufgaben) wird auch die graphische Leistungsfähigkeit des Rechners, die Ausführungsgeschwindigkeit von Skripten, die Netzwerkgeschwindigkeit und die Startzeit von Prozessen gemessen. Die Tests sollten die Frage beantworten, inwieweit sich die Arbeitsgeschwindigkeit eines Arbeitsplatzrechners verändert, wenn im Hintergrund ein JoSchKa-Agent Jobs ausführt (Abb. 4.13).

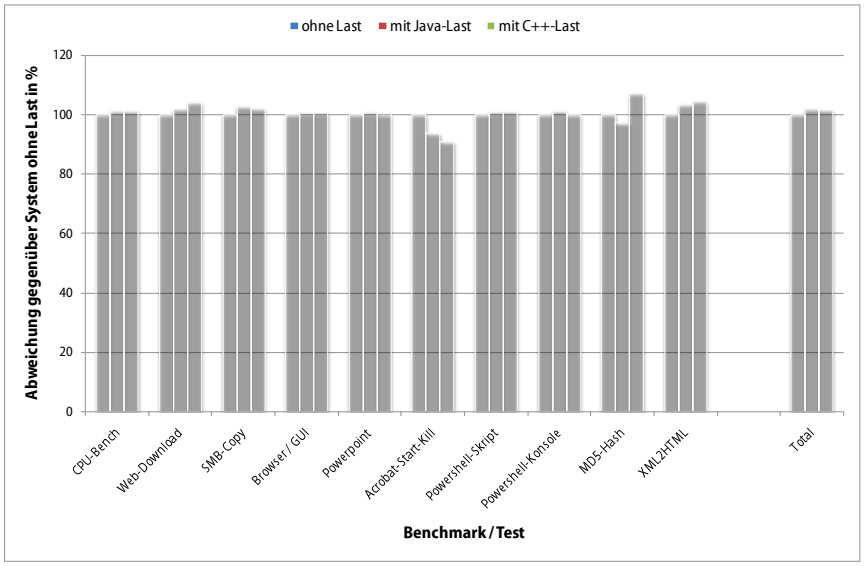


Abb. 4.13 Einfluss von Rechenjobs auf die Systemleistung

Die Abbildung zeigt die Abweichung der Systemperformance bei Hintergrundlast gegenüber dem lastfreien System, dessen Leistungswerte auf 100% normiert wurden. Die einzelnen Testläufe wurden je fünf Mal durchgeführt, der schnellste und der langsamste Messwert wurde jeweils gestrichen, die verbleibenden drei Werte wurden gemittelt und mit dem Normwert verglichen.

Man erkennt deutlich, dass der Einfluss minimal ist. Die Tatsache, dass manche Tests bei Hintergrundlast sogar schneller abliefen, zeigt, dass das Ausführen eines Hintergrundprozesses nicht grundsätzlich Leistung kostet und dass die üblichen Leistungsschwankungen den Einfluss eines Hintergrundjobs überlagern können. Dies gilt selbstverständlich nur, wenn durch den im Hintergrund ausgeführten Rechenjob kein Hauptspeichermangel auftritt. Verbraucht die Hintergrundlast nicht nur CPU-Zeit, sondern viel Hauptspeicher, kann der Einfluss auf die normalen Anwendungen erheblich sein. Aus diesem Grund kann bei einem System, das wenig Hauptspeicher installiert hat, über eine administrativ vorgegebene Plattformbeschreibung (siehe Kapitel 4.5) die Menge der ausgeführten Jobs auf solche eingeschränkt werden, die wenig Hauptspeicher beanspruchen.

Kapitel 5

Zusammenfassung, Praxiserfahrungen und Ausblick

Die in der vorliegenden Arbeit untersuchten Problemstellungen und entwickelten Lösungsansätze beschäftigen sich mit der Frage, auf welche Art und Weise ein Softwaresystem zur Verteilung rechenintensiver Jobs realisiert werden kann, das mit vergleichbar wenig Einarbeitungszeit genutzt werden kann und das es gleichzeitig ermöglicht, vorhandene Standard-PCs mit minimalem administrativem Aufwand in einen Rechenverbund zu integrieren. Ein wesentlicher Aspekt beschäftigte sich mit der Frage, wie sich der auf diese Weise entstehende heterogene Rechnerverbund mit seinen unterschiedlichsten Leistungs- und Zuverlässigkeitscharakteristika optimal mit Arbeit versorgen lässt.

Im Rahmen der vorliegenden Arbeit wurde mit JoSchKa ein System zur Verteilung von rechenintensiven Aufgaben, sogenannten Jobs, auf beliebige Arbeitsplatz-PCs realisiert. Die prinzipielle Architektur des Systems beruht auf der Idee, dass eine zentrale Komponente (Server) die Jobs verwaltet und die auf den Rechenknoten verteilt agierenden Clients (Agenten) autonom bei diesem Server nach Jobs anfragen. Erfolgt eine Anfrage, entscheidet der Server, welcher Job dem anfragenden Agenten zugeteilt wird. Die Agenten laden selbsttätig alle in der Jobbeschreibung spezifizierten Daten vom Server herunter, führen den Job aus und übertragen nach dessen Beendigung die erzeugten Ergebnisdateien zurück zum Server. Dabei kommen ausschließlich Internettechnologien zum Einsatz, die eine lose Kopplung der Komponenten ermöglichen. Ein gemeinsames Dateisystem oder eine Möglichkeit zum Remote-Login auf den Rechenknoten ist nicht notwendig. Somit können diese, im Gegensatz zu den Ausführungsrechnern bei Systemen wie Condor, problemlos hinter NAT-Routern und Firewalls betrieben werden, so dass prinzipiell jeder gewöhnliche Arbeitsplatzrechner als Rechenknoten betrieben werden kann.

Die Rechner, auf die JoSchKa die Jobs verteilt, weisen damit eine potentiell hohe Heterogenität auf, vor allem in Bezug auf die Art und Weise, wie zuverlässig sie jeweils arbeiten. Unter der realistischen Annahme, dass keinerlei Garantie über die vollständige Ausführung eines Jobs besteht, wurden verschiedene neuartige Verteilverfahren implementiert, die darauf basieren, dass zunächst versucht wird, die Verfügbarkeitseigenschaften der einzelnen Rechner zu ermitteln und auf Basis der gemessenen Daten vorherzusagen. Eine wesentliche Eigenschaft von JoSchKa besteht deshalb in der Analyse der Rechenknoten und in der Anpassung der Verteilungsstrategien an eben diese Beobachtungen. Dabei hat sich in verschiedenen Tests die vorhergesagte Betriebszeit eines Rechners bis zum nächsten Ausfall als das sinnvollste Maß erwiesen. Unterscheiden sich die verfügbaren Jobs

der verschiedenen Nutzer signifikant in ihren Anforderungen, werden sie je nach ihrer erwarteten Dauer den vorhandenen Knoten zugeordnet, dass die durch Rechnerausfälle unnütz verbrauchte Rechenkapazität – und damit auch die Energiekosten – verringert werden. Der Scheduler berücksichtigt darüber hinaus grundsätzlich die wichtige Prämisse, Benutzern mit gleichen Anforderungen die gleiche Anzahl an Rechenknoten zur Verfügung zu stellen. Bei keinem mit JoSchKa vergleichbaren Verteilsystem finden sich derartige Ansätze.

JoSchKa stellt nur wenige Bedingungen an die vom Nutzer zu verwendende Programmiersprache. Prinzipiell kann jedes Programm, das ohne Benutzerinteraktivität und ohne Installationsprozedur auf einem Rechenknoten lauffähig ist, mit dem System verteilt werden, was JoSchKa vom auf C/C++ und Fortran fixierten BOINC unterscheidet, bei dem darüberhinaus für jedes parallele Projekt eine separate, aufwändige Serverinfrastruktur bereitgehalten werden muss. Der notwendige Einarbeitungsaufwand des Anwenders von JoSchKa ist damit eher gering, zumal ein leicht und intuitiv zu bedienendes graphisches Frontend zur Verfügung steht und Nutzer nicht auf Kommandozeilenwerkzeuge wie bei Condor oder Alchemi angewiesen sind. Der administrative Aufwand, einen normalen Arbeitsplatz- oder Heim-PC als Rechenknoten in das System zu integrieren, ist, gerade im Vergleich zu Condor, minimal. Zur programmgesteuerten Nutzung verfügt JoSchKa über eine objektorientierte Programmierschnittstelle, die es ermöglicht, flexibel auf die Ergebnisse von Jobs zu reagieren ohne die eigentliche, batchorientierte Arbeitsweise bedenken zu müssen.

Das beschriebene System wurde dabei nicht nur prototypisch, sondern für den alltäglichen Praxiseinsatz realisiert. Dabei wurde neben einer leistungsfähigen und robusten Implementierung auch Wert darauf gelegt, dass ein JoSchKa-Server mit wenig Aufwand aufgesetzt und ein Arbeitsplatzrechner schnell als Rechenknoten in das System integriert werden kann.

Überblicksartig werden im Folgenden zunächst die Betriebserfahrungen in einem normalen PC-Pool beschrieben, sowie der Nutzungsumfang, der während der Entwicklungs-, Test- und bisherigen Betriebsphase anfiel. Aus den Erfahrungen im Praxiseinsatz werden schließlich mögliche Erweiterungen von JoSchKa abgeleitet.

5.1 Einsatz in den Pools

Der Einsatz in den CIP-Pools der Fakultät für Wirtschaftswissenschaften der Universität Karlsruhe (TH)[12] mit handelsüblichen Windows- und Linux-PCs hat gezeigt, dass das beschriebene System einsatzfähig ist und im Praxisgebrauch funktioniert. Es stehen mehr als 100 Rechner der 3 GHz-Klasse mit je 1024 Megabytes Hauptspeicher zur Verfügung, die im Laufe von ca. 18 Monaten etwa 200.000 Rechenjobs aus dem Bereich der naturanalogen Optimierungsverfahren[13] berechneten und damit signifikant bei der Entstehung von Forschungsergebnissen mitwirken konnten. Die damit für wissenschaftliche Zwecke nutzbare Rechenleistung ist beachtlich, während die auf den Rechnern ablaufenden Pro-

[12] http://www.wiwi.uni-karlsruhe.de/ze/cip/

[13] http://www.aifb.uni-karlsruhe.de/Forschungsgruppen/EffAlg/frame_forschung.htm

zesse von den daran arbeitenden Studierenden unbemerkt abliefen. Ebenso können problemlos Büro- oder externe Privatrechner nach Bedarf zum Einsatz kommen, auf denen lediglich das Agentenprogramm aktiv sein muss. Weiterer Voraussetzungen bedarf es nicht. Das System eignet sich demnach sowohl für den Einsatz in homogenen Pool- bzw. Clusterumgebungen als auch für den Einsatz im sogenannten Public Ressource Computing.

Zusätzlich zu den Pool-Rechnern sind einige dedizierte Rechenserver mit verschiedensten Prozessoren, Hauptspeicherausbau und Betriebssystemvarianten in das aktive System integriert. Das in Abschnitt 3.3.4ff beschriebene Verteilverfahren sorgt dafür, dass diese Server, die aufgrund ihres Betriebs im nicht zugänglichen Serverraum sehr zuverlässig arbeiten, bevorzugt die lang laufenden Jobs bearbeiteten. Somit konnten die teilweise höchst unterschiedlichen gleichzeitigen Anforderungen nach Rechenleistung zufriedenstellend erfüllt werden.

Insbesondere für die Diplomanden und das wissenschaftliche Personal bedeutete JoSchKa eine große Arbeitserleichterung. Der Grund liegt unter anderem darin, dass die Studierenden während ihrer Ausbildung überwiegend mit der Programmiersprache Java in Berührung kommen und deswegen ihre Optimierungsprogramme hauptsächlich in dieser Sprache geschrieben sind. Die Großrechneranlagen des Rechenzentrums (HP XC4000 oder HP XC6000) erfordern jedoch eine Programmierung in C/C++ oder Fortran, so dass die Nutzung dieser Rechner meist schon aus technischen Gründen ausscheidet. Die Notwendigkeit, einen Messlauf vielfach parallel ausführen zu müssen, ergibt sich häufig erst im Laufe der Entwicklung bzw. gegen Ende der Arbeit, wenn der Optimierungsalgorithmus getestet wird. Für eine Umstellung des Programms auf Supercomputertechnik (wie die Verwendung von C/C++ und die Parallelisierung mittels spezieller Compiler oder Verwendung von MPI) ist es dann meist zu spät. Die vergleichsweise einfache Bedienung des JoSchKa-Systems ermöglichte es den Studierenden, Jobs zu verteilen, bei denen eine parallele Ausführung zunächst nicht angedacht war.

5.2 Ausblick

Trotz der umfangreichen Funktionalität des vorgestellten Verteilsystems existieren zahlreiche Szenarien, bei denen JoSchKa nicht oder nur unzureichend eingesetzt werden kann. Es kann zwar auf seinem aktuellen Stand durchaus als ausgereift angesehen werden, kann jedoch noch den jeweiligen Anforderungen entsprechend ausgebaut und erweitert werden. Die folgenden Abschnitte beschreiben einige Ansatzpunkte.

5.2.1 Datenaustausch zwischen einzelnen Rechenknoten

Eine zentrale Eigenschaft des JoSchKa-Systems besteht darin, dass die einzelnen Knoten nicht direkt miteinander kommunizieren können. Sie kennen die Adressen der potentiellen Kommunikationspartner nicht, und selbst im dem Falle, dass die Adressen bekannt wären, wäre eine direkte Kommunikation ausgeschlossen. Diese wäre aber notwendig, wenn die einzelnen Knoten nicht nur in sich abgeschlossenen Jobs berechnen, sondern gemeinsam an einem einzelnen Problem arbeiten sollen. Der Grund für die Unmöglich-

keit einer direkten Kommunikation liegt darin, dass der Betrieb eines JoSchKa-Agenten explizit hinter NAT-Netzen oder Firewalls funktioniert und der Entwickler einer parallelen Anwendung nicht davon ausgehen kann, mit anderen Rechnern direkt eine Verbindung herstellen zu können. Zusätzlich stellt sich das Problem, dass eine eventuelle Kommunikation zwecks Datenaustauschs zwischen Rechenknoten immer problemspezifisch ist, also grundsätzlich vom Entwickler der Anwendung initiiert werden müsste. Aufgrund der Netzwerkinfrastruktur, in der JoSchKa eingesetzt wird, scheiden typische Parallelisierungstechniken wie MPI komplett aus. Auch die TCP-Kommunikation über ein selbst definiertes Protokoll ist deshalb nicht durchführbar.

Eine indirekte Kommunikation ist jedoch möglich. Dazu würde ein Server eingerichtet, der Datenpakete oder Dateien von den parallel ausgeführten Rechenprogrammen entgegennimmt und über eine Art schwarzes Brett anderen Knoten zur Verfügung stellt. Die Schnittstelle müsste für Entwickler einfach anzusprechen sein und folgende Basisfunktionen anbieten:

- Ablage von Daten in Form von beliebigen Dateien, sowie eine beliebige Beschreibung derselben,
- Anfrage nach angelegten Daten in Form einer Suchanfrage und
- Download der gefundenen Dateien.

Da diese Schnittstelle vom eigentlichen Rechenprogramm und damit eventuell von vielen verschiedenen Programmiersprachen aus angesprochen werden müsste, kämen voraussichtlich keine Webservices zum Einsatz, sondern reines HTTP oder möglicherweise auch Kombinationen verschiedener Protokolle. Das Standardprotokoll des WWW ist gut geeignet, weil es für viele Programmiersprachen Bibliotheken gibt, mit denen sich leicht HTTP-Anfragen, Datei-Uploads und Downloads formulieren lassen, ohne dass die Protokolldetails selbst implementiert werden müssten.

5.2.2 API-Erweiterungen

Die in Abschnitt 3.7 vorgestellte Programmierschnittstelle ermöglicht es, auf bequeme Art und Weise parallele Anwendungen zu entwickeln, zur Laufzeit Berechnungsergebnisse auszuwerten und entsprechend zu reagieren (z. B. neue Threads zu starten, oder vorhandene abzubrechen). Wie viele andere Programmierschnittstellen hat auch diese die Eigenschaft, nur für eine Programmiersprache oder eine Ablaufumgebung verwendbar zu sein. Durch den Vielsprachenansatz von .NET ist die JoSchKa Thread-API zumindest nicht auf die Sprache C# beschränkt, sondern kann mit allen .NET-fähigen Sprachen (z. B. Delphi oder VB.NET) eingesetzt werden. Sprachen wie Java oder das klassische C/C++ scheiden jedoch aus. Durch die Arbeitsweise der API, den parallelen Code erst zur Laufzeit an den Server zu übertragen, wäre eine Anpassung etwa an Java mit dem Aufwand verbunden, ein Java-Programm bereitzustellen, das dann den mobilen Code in der zur Laufzeit an den Server übertragenen .class-Datei ausführen kann. Für C/C++ könnte der mobile Code in eine DLL verpackt werden, die dann beim Client entsprechend einzubinden wäre.

Da Java umfassende Webservice- und HTTP-Bibliotheken besitzt, würde es wohl kein größeres technisches Problem darstellen, eine Klassenbibliothek zu implementieren, die die Fähigkeiten der JoSchKa-API in der Java-Welt verfügbar macht. Für die Einbindung von C/C++ müsste dagegen ein deutlich größerer Aufwand betrieben werden. Teile der notwendigen Kommunikationsprotokolle wären selbst zu implementieren.

5.2.3 Vertrauensmodell

Wie in Abschnitt 3.5.1 beschrieben erfordert der Betrieb von JoSchKa ein gewisses gegenseitiges Vertrauen: Der Nutzer vertraut dem Betreiber des Rechenknotens, dass die Ergebnisse eines Jobs nicht vor dem Upload verfälscht werden, und der Betreiber eines Rechenknotens vertraut den Nutzern, dass die Jobs den zur Verfügung gestellten Rechner nicht kompromittieren und für andere Zwecke (z. B. Denial-Of-Service-Angriffe auf fremde Dienste oder Einsatz als Email-Spam-Relay) missbrauchen. Der Einsatz von verschiedenen Vertrauensstufen oder Vertrauenszonen könnte hier Abhilfe schaffen:

- Auf dem Server werden verschiedene Vertrauensstufen definiert und zu jeder dieser Stufen ein Schlüssel hinterlegt.
- Vertrauenswürdige Agentenbetreiber erhalten den entsprechenden Schlüssel und statten ihr Agentenprogramm damit aus.
- Der Entwickler definiert bei seinen Jobs einen bestimmten Vertrauenslevel, den der Agent, auf dem der Job ausgeführt werden soll, erfüllen muss.
- Stellt der Agent eine Anfrage nach einem Job, wird die Anfrage oder ein Teil davon mit diesem Schlüssel verschlüsselt. Der Server kann dann dem Level des Agenten entsprechende Jobs zuteilen.

Die umgekehrte Vertrauensrichtung (der Betreiber eines Agenten vertraut dem Entwickler) könnte man mit digitalen Signaturen erreichen:

- Vertrauenswürdige Entwickler erhalten ein Zertifikat, mit dem sie die vom Agenten ausgeführten Dateien signieren müssen.
- Der Betreiber eines Rechenknotens könnte dann den Agenten so konfigurieren, dass dieser nur Jobs ausführt, bei denen allen Dateien eine gültige Signatur beiliegt.

5.2.4 Abrechnungsmodell

JoSchKa entstand unter der Voraussetzung, ein für alle Nutzer kostenfreies, zunächst prototypisch realisiertes System anzubieten. Jedoch ist es auch denkbar, sich die angebotene Rechenleistung vergüten zu lassen. Dazu wäre es notwendig, für jeden Benutzer zu erfassen, wie viel Rechenleistung für ihn erbracht wurde und wie sich diese genutzte Rechenleistung auf die einzelnen Knoten (und damit deren Besitzer) verteilt. Auf diese Weise ließe sich beispielsweise bestimmen, wie viel Stromkosten die Rechenjobs bei den Betreibern der Rechenknoten verursacht haben und wie diese Kosten auf die jeweiligen Nutzer umzulegen wären.

Ebenso wäre es denkbar, die verfügbare Rechenleistung zu verkaufen: Je mehr ein Kunde vorab bezahlt, desto mehr (leistungsfähige, zuverlässige) Rechenknoten würden für ihn arbeiten.

5.3 Fazit

Abschließend ist festzuhalten, dass das im Rahmen der vorliegenden Dissertation realisierte JoSchKa – unter der Vielzahl verschiedener existierender Systeme zum verteilten Rechnen – neuartige Konzepte zur Nutzung und Bereitstellung eines Jobverteilungssystems sowie der darin eingesetzten Schedulingalgorithmen vorschlägt.

Darüber hinaus hat sich JoSchKa im Praxiseinsatz der Universität Karlsruhe (TH) dauerhaft bewährt.

Literatur

[Alc07] Alchemi .NET Grid Computing Framework. (2007). (Melbourne, University of) Abgerufen am 10. Oktober 2007 von http://www.alchemi.net/

[And04] Anderson, D. (2004). BOINC: A System for Public Resource Computing and Storage. 5th IEEE/ACM International Workshop on Grid Computing, (pp. 365–372). Pittsburg, PA.

[ACKL+02] Anderson, D., Cobb, J., Korpela, E., Lebofsky, M., & D., W. (2002, November). SETI@home: An Experiment in Public Resource Computing. *Communications of the ACM , 45* (11), pp. 56–61.

[Bar05] Barth, W. (2005). *Nagios. System- und Netzwerk-Monitoring* (1 Ausg.). Open Source Press.

[BHB04] Bode, B., Hill, J., & Benjegerdes, T. (July 2004). Cluster Interconnect Overview. *Proceedings of the USENIX Extreme Linux Technical Conference*, (pp. 211–218). Boston, Massachusetts.

[BKCB+06] Byun, E., Kim, H., Choi, S., Baik, M., Goo, S., Gil, J., et al. (2006). Advanced Stochastic Host State Modeling to Reliable Computation in Global Computing Environment. *Embedded and Ubiquitous Computing. LNCS 4096*, S. 183–192. Seoul, Korea: Springer Berlin Heidelberg.

[BNW03] Brevik, J., Nurmi, D., & Wolski, R. (2003). *Quantifying Machine Availability in Networked and Desktop Grid Systems.* Technical Report CS2003-37, Dept. of Computer Science and Engineering, University of California at Santa Barbara.

[Boi07] *BOINC Homepage*. (2007). (California, University of) Abgerufen am 2. Oktober 2007 von http://boinc.berkeley.edu/

[BoSc07] Bonn, M., & Schmeck, H. (2007). Strategien zur Effizienzsteigernden Verteilung von Rechenjobs auf unzuverlässige Knoten im JoSchKa-System. *PARS Mitteilungen 2007*, (S. 100–113). 21. PARS Workshop, Gesellschaft für Informatik, Hamburg.

[Bro96] Brown, M. (1996). FastCGI: A High-Performance Gateway Interface. *Fifth International World Wide Web Conference*. Paris, France.

[BSV03] Bhagwan, R., Savage, S., & Voelker, G. (2003). Understanding Availability. *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*. Berkeley, CA, USA.

[BTS05] Bonn, M., Toussaint, F., & Schmeck, H. (2005). JoSchKa: Job-Scheduling in heterogenen Systemen. In E. Maehle (Hrsg.), *PARS Mitteilungen 2005*, (S. 99–106). 20. PARS Workshop, Gesellschaft für Informatik, Lübeck.

[CCEB03] Chien, A., Calder, B., Elbert, S., & Bhatia, K. (2003, May). Entropia: Architecture and Performance on an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing , 63* (5).

[CIFS02] *Common Internet File System (CIFS) Technical Reference*. (2002). (Storage Networking Industry Association) Abgerufen am 10. Dezember 2007 von http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf

[Cli07] *Climaprediction.net*. (2007). Abgerufen am 2. Oktober 2007 von http://www.climateprediction.net/

[CLZB00] Casanova, H., Legrand, A., Zagrorodnov, D., & Bernen, F. (2000). Heuristics for Parameter Sweep Applications in Grid Environments. *Ninth IEEE Heterogenous Computing Workshop*, (pp. 349–363).

[Con07] *Condor Project Homepage*. (2007). (University of Wisconsin-Madison) Abgerufen am 6. September 2007 von http://www.cs.wisc.edu/condor/

[CPPM+07] Cera, M., Pezzi, G., Pilla, M., Maillard, N., & Navaux, P. (2007). Scheduling Dynamically spawned Processes in MPI-2. In U. Schwiegelshohn (Hrsg.), *12th International Workshop JSSPP 2006. LNCS 4376*, S. 33–46. Saint-Malo, France: Springer Verlag Berlin Heidelberg 2007.

[DJMZ05] Dostal, W., Jeckle, M., Melzer, I., & Zengler, B. (2005). *Service-orientierte Architekturen mit Web Services*. Spektrum Akademischer Verlag.

[DMS99] Dongarra, J., Meuer, H., & Strohmaier, E. (June 1999). Top500 Supercomputer Sites 13th Edition. *Supercomputer 1999.* Mannheim.

[Ein07] *Einstein@home Homepage*. (2007). (Collaboration, LIGO Scientific) Abgerufen am 2. Oktober 2007 von http://einstein.phys.uwm.edu/

[ErSn01] Erwin, D., & Snelling, D. (2001). UNICORE: A Grid Computing Environment. *Proceedings of 7th International Conference Euro-Par* (S. 825–834). Manchester, UK: Springer LNCS 2150.

[EtTs05] Etsion, Y., & Tsafrir, D. (2005). *A Short Survey of Commercial Cluster Batch Schedulers.* School of Computer Science and Engineering, The Hebrew University of Jerusalem.

[FCFF+05] Foster, I., Czajkowski, K., Ferguson, D., Frey, J., Graham, S., Maguire, T., et al. (2005, March). Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF. *Proceedings of the IEEE , 93* (3), pp. 604–612.

[FoKe99] Foster, I., & Kesselman, C. (1999). *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc.

[Fos02] Foster, I. (20. July 2002). *What is the Grid? A three Point Checklist.* Abgerufen am 14. Januar 2008 von http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf

[Fos06] Foster, I. (2006). Globus Toolkit Version 4: Software for Service-Oriented Systems. *IFIP International Conference on Network and Parallel Computing* (pp. 2–13). Springer-Verlag LNCS 3779.

[FPFP06] Frachtenberg, E., Petrini, F., Fernández, J., & Pakin, S. (2006, December). STORM: Scalable Resource Management for Large-Scale Parallel Computers. *IEEE Transactions on Computers , 55* (12), pp. 1572–1587.

[FTFL+02] Frey, J., Tannenbaum, T., Foster, I., Livny, M., & Tuecke, S. (2002). Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing , 5*, pp. 237–246.

[GBus07] *The Gridbus Project*. (2007). (The University of Melbourne) Abgerufen am 9. Oktober 2007 von http://www.gridbus.org/

[GBDW+94] Geist, A., Beguelin, A., Dongarra, J., Weicheng, J., Manchek, R., & Sungeram, V. (1994). *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press .

[Gel85] Gelernter, D. (1985). Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems , 7* (1), pp. 80–112.

[GNFC00] Germain, C., Neri, V., Fedak, G., & Cappello, F. (2000). XtremWeb: Building an experimental Platform for Global Computing. *Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000).* Bangalore, India.

[GRV04] Giersch, A., Robert, Y., & Vivien, F. (2004). Scheduling Tasks Sharing Files on Heterogenous Master-Slave Platforms. *12th IEEE Euromico Workshop Parallel and Distributed Network-Based Processing.*

[HHL07] He, Y., Hsu, W., & Leiserson, C. E. (2007). Provably Efficient Two-Level Adaptive Scheduling. In U. Schwiegelshohn (Hrsg.), *12th International Workshop JSSPP 2006. LNCS 4376*, S. 1–32. Saint-Malo, France: Springer Verlag Berlin Heidelberg 2007.

[HuSc04] Huttary, R., & Schäpers, A. (2004). Verteilte Welten: Mit .NET Remoting prozessübergreifend kommunizieren. *c't* (13), S. 214–215.

[Inf07] *Infiniband*. (2007). (Infiniband Trade Association) Abgerufen am 9. Januar 2008 von http://www.infinibandta.org

[Int08] *Intel Xeon Processor HPC Benchmarks*. (2008). (Intel Corporation) Abgerufen am 13. Mai 2008 von http://www.intel.com/performance/server/xeon/hpcapp.htm

[KCBW02] Kondo, D., Casanova, H., Berman, F., & Wing, E. (2002). Models and Scheduling Mechanisms for Global Computing Applications. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '02)* (p. 216). Fort Lauderdale, Florida, USA: IEEE Computer Society Washington, DC, USA.

[KEF01] Kavas, A., Er-El, D., & Feitelson, D. (Februar 2001). Using Multicast to Preload Jobs on the ParPar Cluster. *Parallel Computing , 27* (3), S. 315–327.

[KuWö02] Kuschke, M., & Wölfel, L. (2002). *Web Services kompakt.* Spektrum Akadademischer Verlag, Heidelberg Berlin.

[Lau06] Lau, O. (2006). Schnellimbiss – Cluster programmieren mit dem Message Passing Interface (MPI). *c't* (26), S. 224–231.

[LBRV05] Luther, A., Buyya, R., Ranjan, R., & Venugopal, S. (2005). Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. In L. Y. Guo, *High Performance Computing: Paradigm and Infrastructure.* New Jersey, USA: Wiley Press.

[LHC08] *Worldwide LHC Computing Grid*. (2008). Abgerufen am 14. Januar 2008 von http://lcg.web.cern.ch/LCG/index.html

[LLM88] Litzkow, M., Livny, M., & Mutka, M. (1988). Condor – a Hunter of idle Workstations. *8th International Conference on Distributed Computing Systems*, (pp. 104–111).

[MASH+99] Maheswaran, M., Ali, S., Siegel, H., Hensgen, D., & Freund, R. (1999). Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Proceedings of the Eighth Heterogeneous Computing Workshop* (S. 30). San Juan, Puerto Rico: IEEE Computer Society.

[MiNo06] Mickens, J., & Noble, B. (2006). Exploiting Availability Prediction in Distributed Systems. *NSDI '06: 3rd Symposium on Networked Systems Design & Implementation*, (S. 73–86). San Jose, CA, USA.

[MCC03] Massie, M., Chun, B., & Culler, D. (2003). *The Ganglia Distributed Monitoring System: Design, Implementation And Experience.* Technical Report, University of California, Berkeley.

[MPI07] *Message Passing Interface Forum*. (2007). (Message Passing Interface Forum) Abgerufen am 19. Dezember 2007 von http://www.mpi-forum.org/

[Myr07] *Myrinet 2000*. (2007). (Myricom, Inc.) Abgerufen am 9. Januar 2008 von http://www.myricom.com/myrinet/

[NeKe07] Neuroth, H., & Kerzel, M. (2007). *Die D-Grid Initiative.* (W. Gentzsch, Hrsg.) Universitätsverlag Göttingen.

[Qsn04] *QsNet High Performance Interconnect*. (2004). (Quadrics) Abgerufen am 9. Januar 2008 von http://www.quadrics.com/Quadrics/QuadricsHome.nsf/DisplayPages/3A912204F260613680256DD9005122C7

[RFC791] *Internet Protocol*. (1981). (Information Sciences Institute, University of Southern California) Abgerufen am 5. Dezember 2007 von http://tools.ietf.org/html/rfc791

[RFC793] *Transmission Control Protocol*. (1981). (Information Sciences Institute, University of Southern California) Abgerufen am 5. Dezember 2007 von http://tools.ietf.org/html/rfc793

[RFC959] *File Transfer Protocol (FTP)*. (1985). (The Internet Socienty) Abgerufen am 6. Dezember 2007 von http://tools.ietf.org/html/rfc959

[RFC1034] *Domain Names – Concepts and Facilities*. (1987). (The Internet Society) Abgerufen am 5. Dezember 2007 von http://tools.ietf.org/html/rfc1034

[RFC1631] *The IP Network Address Translator (NAT)*. (1994). (The Internet Society) Abgerufen am 5. Dezember 2007 von http://tools.ietf.org/html/rfc1631

[RFC2246] *The TLS Protocol Version 1.0*. (1999). (The Internet Society) Abgerufen am 12. Dezember 2007 von http://tools.ietf.org/html/rfc2246

[RFC2616] *Hypertext Transfer Protocol – HTTP/1.1*. (1999). (The Internet Society) Abgerufen am 5. Dezember 2007 von http://tools.ietf.org/html/rfc2616

[RFC2617] *HTTP Authentication: Basic and Digest Access Authentication*. (1999). (The Internet Society) Abgerufen am 5. Dezember 2007 von http://tools.ietf.org/html/rfc2617

[RFC3986] *Uniform Resource Identifier (URI): Generic Syntax*. (2005). (The Internet Society) Abgerufen am 5. Dezember 2007 von http://tools.ietf.org/html/rfc3986

[RFC4251] *The Secure Shell (SSH) Protocol Architecture*. (2006). (The Internet Society) Abgerufen am 2007 von http://tools.ietf.org/html/rfc4251

[RFC4918] *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*. (2007). (The IETF Trust) Abgerufen am 6. Dezember 2007 von http://tools.ietf.org/html/rfc4918

[RiMn08] Richter, U., & Mnif, M. (2008). Learning to Control the Emergent Behaviour of a Multi-agent System. *Proceedings of the 2008 Workshop on Adaptive Learning Agents and Multi-Agent Systems at AAMAS 2008.*

[SCCK08a] *Manuals InstitutsCluster*. (2008). (SCC Karlsruhe) Abgerufen am 5. Mai 2008 von http://www.rz.uni-karlsruhe.de/ssck/ic-manuals.php

[SCCK08b] *Projektanträge Landeshöchstleistungsrechner HP XC*. (2008). (SCC Karlsruhe) Abgerufen am 5. Mai 2008 von http://www.rz.uni-karlsruhe.de/ssck/proposals.php

[Sch06a] Schwichtenberg, H. (2006). *Microsoft .NET 2.0 Crashkurs.* Microsoft Press, Unterschleißheim.

[Sch06b] Schwichtenberg, H. (2006). *ASP.NET 2.0 mit Visual C# 2005.* Microsoft Press, Unterschleißheim.

[Sch07] Schmidt, C. (2007). *Evolutionary Computation in Stochastic Environments.* Karlsruhe: Universitätsverlag.

[SOAP07] *SOAP Version 1.2 Part 1: Messaging Framework*. (2007). (W3C) Abgerufen am 2007. Dezember 11 von http://www.w3.org/TR/soap12

[Str04] Strobel, C. (2004). *Web-Technologien in E-Commerce-Systemen.* Oldenbourg Wissenschaftsverlag GmbH, München.

[Tan97] Tanenbaum, A. S. (1997). *Computernetzwerke.* Prentice Hall Verlag GmbH, München.

[TavS03] Tanenbaum, A. S., & van Steen, M. (2003). *Verteilte Systeme – Grundlagen und Paradigmen.* Pearson Education Imprint GmbH, München.

[TEF07] Tsafrir, D., Etison, Y., & Feitelson, D. (2007, June). Backfilling Using System-Generated Predictions rather than User Runtime Estimates. *IEEE Transactions on Parallel and Disttributed Systems , 18* (6), pp. 789–803.

[Tro02] Troelsen, A. (2002). *C# und die .NET-Plattform.* mitp Verlag, Bonn.

[TWML01] Tannenbaum, T., Wright, D., Miller, K., & Livny, M. (2001). Condor – A Distributed Job Scheduler. In T. Sterling, *Beowulf Cluster Computing with Linux.* MIT Press.

[UDDI04] *OASIS UDDI Specifications TC – Committee Specifications*. (2004). (OASIS) Abgerufen am 11. Dezember 2007 von http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

[VySu07] Vyas, D., & Subhlok, J. (2007). Volunteer Computing on Clusters. In U. Schwiegelshohn (Ed.), *12th International Workshop JSSPP 2006. LNCS 4376*, pp. 161–175. Saint-Malo, France: Springer Verlag Berlin Heidelberg 2007.

[WPD01] Whaley, C. R., Petitet, A., & Dongarra, J. (2001). Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing , 27* (1–2), pp. 3–35.

[WSDL01] *Web Services Description Language (WSDL) 1.1*. (2001). (W3C) Abgerufen am 11. Dezember 2007 von http://www.w3.org/TR/wsdl

ISBN: 978-3-86644-288-7

www.uvka.de