JOSCHKA: Job-Scheduling in heterogenen Systemen mit Hilfe von Webservices

Matthias Bonn, Frederic Toussaint, Hartmut Schmeck

Institut AIFB, Universität Karlsruhe (TH)
Englerstr. 11, 76131 Karlsruhe
bonn@aifb.uni-karlsruhe.de
toussaint@wiwi.uni-karlsruhe.de
schmeck@aifb.uni-karlsruhe.de

Abstract. JOSCHKA (Job Scheduling Karlsruhe) ist ein System zum Verteilen von Rechenjobs auf verschiedene voneinander unabhängige Rechensysteme. Es handelt sich dabei um ein Client-Server-basiertes Verteilsystem, das sich durch folgende Eigenschaften auszeichnet: Es unterstützt heterogene Systeme, Kommunikation und Datenaustausch erfolgen dabei ausschließlich mit Webtechnologien. Dabei ist das System generisch, für viele unterschiedliche Aufgaben einsetzbar, für Entwickler leicht zu benutzen und erfordert keinerlei Installationsaufwand bei den Clients. Im Folgenden werden die Arbeitsweise des Systems und die zugrunde liegende Architektur beschrieben. Am Schluss folgen Praxiserfahrungen.

1 Aufgabenverteilung im Netz

Seit Jahren steigt die verfügbare Rechenleistung von Personal Computern kontinuierlich an. Ein normaler Arbeitsplatzrechner hat heute die Leistung, die vor wenigen Jahren ausschließlich Großrechneranlagen vorbehalten war. Diese Rechenkapazität wird den Rechnern allerdings meist nicht abverlangt: Eine moderne CPU befindet sich – gerade wenn sie in einem Büro- oder Heimrechner eingesetzt wird – überwiegend im Leerlauf. Im Zuge der fortschreitenden Verbreitung des Internets seit dem Ende der 1990er Jahre existieren einige Projekte, die diese brachliegende Rechenleistung für wissenschaftliche Zwecke nutzen. Neben dem bekannten SETI@home (das inzwischen unter Einsatz der BOINC-Technologie [2] weitergeführt wird), existieren noch einige weitere dieser „Public-Ressource Computing"-Projekte. Als Beispiel sei hier zetaGrid [3] genannt. Neben diesen meist auf eine bestimmte Aufgabe spezialisierten Projekten existieren die Cluster-Systeme, die flexible Verfahren zur Verteilung von vielfältigen Problemstellungen auf homogenen Rechnernetzen bereitstellen [4, 5, 6]. Sie eignen sich allerdings nicht notwendigerweise dazu, brachliegende Rechenleistung auf normalen Arbeitsplatz- oder gar Internet-PCs zu nutzen.

Dieser Beitrag beschreibt ein System, das sich für verschiedenste parallele Problemstellungen einsetzen lässt, dabei aber nicht auf homogene, ausschließlich zum verteilten Rechnen eingesetzte Rechner beschränkt ist. Dies wird durch den Einsatz von Mechanismen des „Public Computing" erreicht.

1.1 Rechenjobverteilung im homogenen LAN: Probleme und Ziele

Bisher werden Rechenjobs überwiegend durch Einsatz von Fernzugriffs- und Netzwerkdateisystemtechnologien verteilt: Per Remote-Login auf einem Clusterrechner wird der eigentliche Arbeitsbefehl gestartet, dies wird in der Regel mit diversen Skriptsprachen wie Python oder Perl automatisiert. Die Quell- und Ergebnisdateien eines Rechenjobs müssen sich dabei auf einem für alle Rechenknoten gemeinsamen Dateisystem wie NFS oder CIFS befinden. Es gibt eine Reihe von freien (z.B. Cactus [5] oder Condor [6]) und kommerziellen Produkten (z.B. PBSpro [4], das im Forschungszentrum Karlsruhe zur Auswertung von Teilchenbeschleuniger-Experimenten genutzt wird [7] und auf dem Globus-Toolkit [8] basiert), die auf diesem Prinzip arbeiten. Ein Nachteil hängt mit der Homogenität des LAN zusammen: Die einzelnen Knoten müssen in der Lage sein, eine z.B. per NFS exportierte Dateisystemressource zu „mounten" und brauchen einen Remote-Zugang (z.B. per SSH-Shell). Dies ist jedoch nur in der Unixwelt üblich und im (Firewall-gesicherten) Internet nicht praktikabel. Condor etwa richtet auf dem Rechenknoten eine dynamische Anzahl von Netzwerkdiensten ein („offene TCP-Ports"), was den Einsatz dieses Systems über Netzgrenzen hinweg unmöglich macht. Weiter nachteilig wirkt sich aus, dass Clustersysteme oft gewisse Voraussetzungen bezüglich der installierten Software haben: Cactus z.B. benötigt unter anderem den Cygwin Emulator, einen Perl-Interpreter sowie MPI und diverse Compiler auf den Zielsystemen. Systeme wie BOINC erlauben zwar den Einsatz in NAT-gesicherten Netzen und benötigen keine Netzwerkdienste auf den Rechenknoten, sind jedoch hoch spezialisiert und verlangen vom Entwickler, gegen eine bestimmte Schnittstelle oder Klassenbibliothek zu programmieren. Es können hier also keine beliebigen, bereits vorhandenen Programme benutzt werden. Will man ein flexibles, weltweit verfügbares „Grid"[1]-System zum verteilten Rechnen aufbauen, müssen andere Technologien verwendet werden. Die Ziele können wie folgt zusammengefasst werden:

- 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,
- verbesserte Kontrolle über einzelne Jobs und Rechenknoten und
- faire Verteilung der Jobs aller beteiligten Entwickler.

Der in diesem Beitrag beschriebene Ansatz folgt der Idee, dass ein zentraler Server die einzelnen Jobs verwaltet und ein Rechenknoten selbsttätig beim Server anfragt, ob es etwas zu bearbeiten gibt. Der gesamte Rechenjob wird daraufhin vom Server heruntergeladen, ausgeführt und danach das Ergebnis wieder zum Server übertragen (Abbildung 1 links).

1.2 Rechenjobverteilung mit Webservices

Die Grundidee des hier beschriebenen Systems liegt darin, dass ein Server existiert, der ein Portfolio an Rechenjobs bekommt, aus denen er einen geeigneten auswählt sobald ein Client anfragt. Dazu müssen

---

[1] Der Begriff „Grid" im Zusammenhang mit Rechnernetzen stammt von I. Foster und C. Kesselman [1]: Sie bildeten den Begriff in Analogie zum Stromnetz („Grid"), bei dem man Ressourcen (Strom bzw. Rechenleistung) erhält ohne sich um die Herkunft Gedanken machen zu müssen.

sich die einzelnen Jobs ausreichend gut selbst beschreiben. Der Client (im Folgenden auch „Agent“ genannt) spezifiziert beim Anfragen genau, welche Art von Jobs er ausführen kann (z.B. „Gib mir etwas von Benutzer X“) oder auf welcher Plattform die Jobs laufen müssen (z.B. „Gib mir keine Jobs die Java benötigen“). Weiterhin sollte ein Agent auf seiner Plattform beliebige Kommandos ausführen (also Programme oder Skripte starten), beliebige Dateien von und zu einem Server übertragen und dabei völlig autonom agieren können. Die zwischen Client und Server ausgetauschten Status- und Steuerinformationen werden mit Webservice-Technologien übertragen[2]. Der Server besitzt eine SOAP-Schnittstelle (*Simple Object Access Protocol*) [10], Client und Server kommunizieren nur für kurze Zeit miteinander und der Datenaustausch wird dabei grundsätzlich vom Client initiiert. Es sind keine Netzwerkdienste auf dem Clientrechner notwendig. Abbildung 1 rechts veranschaulicht, welche Daten in welcher Reihenfolge zwischen den beteiligten Rechnern fließen.

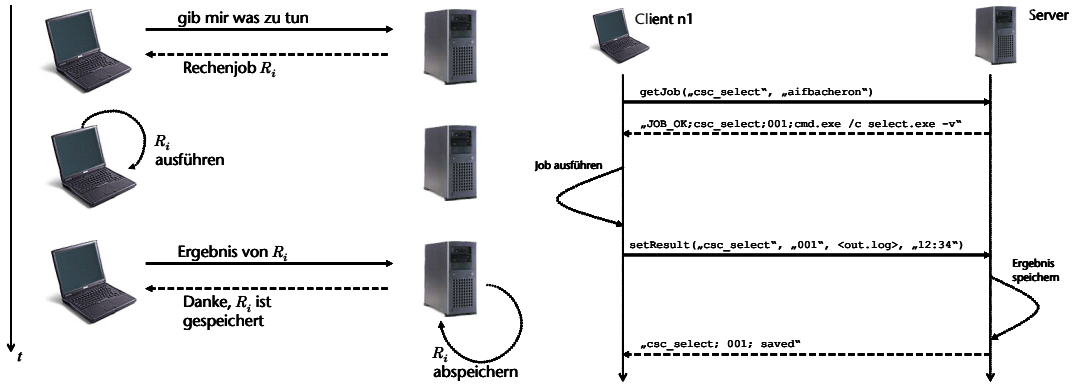

Abbildung 1: Client-gesteuerte Jobverteilung (links), Verteilungsprinzip mit Webservices (rechts)

Im Beispiel erfragt der Client „aifbacheron“ beim Server einen Job vom Typ „csc_select“. Die Antwort des Servers besteht neben Statusinformationen aus einer Befehlszeile „cmd.exe /c select.exe –v“, die der Client ausführt. Während der Ausführung erhält der Server periodisch Statusinformationen vom Client und nach Beendigung der Jobs in einem weiteren Webservice-Aufruf das Endergebnis.

Im Folgenden wird beschrieben, welche Anforderungen an die einzelnen Komponenten gestellt werden, wie sie aufgebaut sein müssen und welche Datenmodelle zum Einsatz kommen. Des Weiteren wird die Serverschnittstelle an ausgewählten Beispielen erläutert. Zusätzlich wird die Arbeitsweise der einzelnen Komponenten beschrieben.

[2] Ein Webservice funktioniert wie ein entfernter Prozeduraufruf, bei dem Funktionsname, Methodenparameter und Rückgabewert in XML serialisiert und (meist) per HTTP übertragen werden.

2 Systementwurf

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

- eine Klassifizierung, die den Job einem Benutzer und innerhalb des Benutzers einem Projekt oder Typ zuordnet,
- eine innerhalb des Portfolios eindeutige Identifikationsnummer,
- das vom Client auszuführende Kommando,
- eine Statusbeschreibung (frei, laufend, fertig…),
- eine Liste von Quell- und Ergebnisdateien und
- statistische Daten über verbrauchte Rechenzeit sowie den ausführenden Knoten.

Die folgende Tabelle zeigt exemplarisch, wie eine solche Jobdatenbasis aufgebaut ist:

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	1:23:45.5
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

Tabelle 1: Jobdatenbasis (vereinfacht)

Die (Tuple-Space-)Daten [9] können z.B. in einem RDBMS (*Relational Database Management System*) oder als XML-Dokument gespeichert werden. Die einzelnen Datenfelder werden als Aufrufparameter oder Rückgabewert der einzelnen SOAP-Funktionen des Webservices realisiert. Der Server besitzt eine per WSDL (*Web Service Description Language*) beschriebene Schnittstelle zur Kommunikation mit den Clients. Hinzu kommen ein Datenhaltungsteil, der die einzelnen Jobs verwaltet, und verschiedene Managementfunktionen, die der Benutzer- und Clientverwaltung dienen.

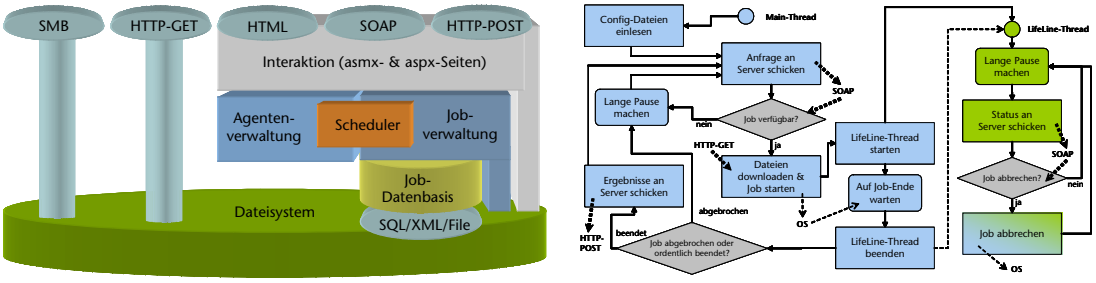

Abbildung 2: Architektur Server (links), Entwurf Agent (rechts)

Die einzelnen Serverkomponenten sowie ihre primären Funktionen können Abbildung 2 links entnommen werden. Die wichtigste Aufgabe übernimmt dabei die Jobverwaltung. Diese Schicht realisiert die Abbildung der SOAP-Anfragen auf die interne Jobrepräsentation und die persistente Speicherung dieser Daten (sofern kein DBMS zum Einsatz kommt). Eine Vielzahl unterschiedlicher Funktionen wird dabei realisiert: Einfügen von Jobs in die Datenbasis, Auswählen eines Jobs bei Client-Anfrage, Löschen von Jobs, Bestätigen einer erfolgreichen Ausführung sowie diverse Fehlererkennungsmechanismen, die dafür sorgen, dass die Jobs eines ausgefallenen Clients wieder für andere freigegeben werden. Ein prob-

lemloser Parallelzugriff auf den Datenbestand und eine konsistente Speicherung der Jobdaten wird gewährleistet.

Die Aufgaben des Clients (oder „Agenten“) lassen sich wie folgt charakterisieren:

1) Einlesen einer Konfigurationsdatei oder Kommandozeilenparametern,
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 Statusmeldungen an den Server schicken und dabei auf das Job-Ende warten,
5) nach erfolgreicher Beendigung des Jobs Ergebnisdatei(en) zum Server hochladen,
6) gegebenenfalls eine kurze Pause machen und von vorne (2) beginnen.

Dieser Ablauf ist rechts in Abbildung 2 dargestellt. Man erkennt, dass auch 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. Ein wichtiger Aspekt in diesem Zusammenhang ist die Prozesspriorität: Ein laufender Rechenjob darf sich auf einem Knoten nur als Hintergrundlast bemerkbar machen und die Arbeit des interaktiv am Rechner arbeitenden Benutzers nicht stören oder beeinträchtigen. Deswegen startet der Agent sämtliche Jobs immer mit der niedrigsten Prozesspriorität (unter Windows 2000, Windows XP und deren Servervarianten „niedrig“, unter Linux wird das Befehlspräfix „nice 19“ benutzt).

3 Realisierung

Der tatsächlich realisierte Prototyp umfasst weitere Funktionen, die im obigen Abschnitt nicht dargestellt wurden. Während der Entwicklung wurden mehrere zusätzliche Funktionalitäten identifiziert, die Server, Agent und Managementprogramme erfüllen müssen. Bei der Implementierung wurde die Sprache C# verwendet die auf Microsofts .NET-Framework [11] basiert. Der Server setzt einen ASP.NET Application Server voraus, der vom MS Internet Information Server IIS bereitgestellt wird. Die Clients benötigen ebenfalls das .NET-Framework, sind ohne GUI auch unter Linux lauffähig, sofern dort die Mono-Laufzeitumgebung [12] vorhanden ist (getestet auf Debian Linux mit Mono 1.1).

3.1 Server

Der Server besitzt vier Schnittstellen (siehe auch Abbildung 2 links), die unterschiedlichen Zwecken dienen und serverseitig unterschiedlich realisiert sind:

Webservice-Schnittstelle: Sie dient dem Austausch von Status- und Steuerinformationen zwischen Agent und Server bzw. zwischen Management-Werkzeugen und Server und bietet die folgenden Funktionen: Verwaltung der einzelnen Jobdaten, Löschen und Hinzufügen 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 Ausfüh-

rung sowie deren Abschluss. Angesprochen wird die Schnittstelle mit dem Webservice-Protokoll SOAP, als Transportprotokoll kommt HTTP zum Einsatz.

Upload-Schnittstelle: Da eine Übertragung von Dateien 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 als klassischen HTTP-Upload ermöglicht. Die Daten werden binär im POST-Teil einer Anfrage übertragen.

Download-Schnittstelle: Sie dient dem effizienten Download der auszuführenden Dateien durch den Agenten. Verwendet wird ein normaler Web-Download mit HTTP-GET.

Quelldateien- und Ergebnisschnittstelle: Da der Entwickler einer parallelen Grid-Anwendung Quelldateien am Server hinterlegen muss und seine Ergebnisdateien letztlich auch erhalten soll, wurde eine SMB-Schnittstelle (*Server Message Block*) eingerichtet, durch die es wesentlich bequemer möglich ist, mit einem einzelnen Arbeitsschritt viele Dateien zu übertragen, als das mit HTTP möglich wäre. Es ist ebenso denkbar, diese Schnittstelle mit FTP, SCP oder WebDAV zu realisieren. Sie wird nur vom Entwickler benötigt und kommt ausschließlich vor und nach der Ausführung der Anwendung zum Einsatz. Während der Jobverteilung und der Ausführung wird sie nicht benötigt. Unter Windows wird diese Schnittstelle durch Einrichten eines Netzlaufwerks angesprochen, unter Unix-Systemen nutzt man das Programm *smbmount*.

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 geeigneten Job und schickt als Antwort die Beschreibung desselben.
3) Der Agent lädt über die Download-Schnittstelle die in der Jobbeschreibung spezifizierten Dateien herunter.
4) Der Agent startet die im Job angegebene Kommandozeile 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 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.

Die während der Ausführung regelmäßig ausgetauschten Zustandsdaten sorgen dafür, dass der Server immer darüber informiert ist, welcher Agent gerade 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 freigege-

ben. Dieses Verfahren stellt einen sicheren Betrieb auch mit stark schwankender und dynamischer Anzahl von unzuverlässigen Rechenknoten sicher.

Ein einzelner Rechenjob wird im Server als 13-Tupel repräsentiert, das intern in einer Tabellenstruktur verwaltet wird. Wie aus den Datenstrukturen aus Tabelle 1 ersichtlich ist, besitzt jeder Job neben der Beschreibung der I/O-Dateien, des Start-Befehls und Verwaltungsinformationen ein weiteres Datenfeld, das die Benutzerzuordnung und innerhalb eines Benutzers die Jobklasse spezifiziert. Zusammen bilden diese beiden Informationen den *JobType*, welcher dem folgenden Zweck dient: Die einzelnen Namensbestandteile des Type werden dazu verwendet, die URL zu erzeugen, die der Agent beim Download der Eingabedateien aufruft, und die Verzeichnisse festzulegen, in denen der Benutzer seine Eingabedateien hinterlegen muss. Der JobType wird ebenso dazu verwendet, die einzelnen Jobs der verschiedenen Typen möglichst gleichmäßig und fair an die Agenten zu verteilen. Der Scheduler versucht allerdings zusätzlich, die Jobs neu hinzugekommener Typen möglichst schnell zu verteilen, damit ein neuer Benutzer schnell erste Ergebnisse erhält. Weiterhin werden Typen, die kurz vor Abschluss stehen beschleunigt, damit ein Benutzer, der fast fertig ist, nicht „verhungert". Ebenso ist es Aufgabe des Schedulers, die Jobs möglichst so an die Agenten zu verteilen, dass lang laufende Jobs an schnelle, und kurz laufende Jobs an leistungsschwächere Clients ausgeliefert werden. Da vor der Ausführung eines Jobs jedoch nicht abzusehen ist, wie lange dessen Bearbeitung dauert, wird versucht, die Laufzeit vorherzusagen. Dazu wird der Laufzeit-Durchschnitt aller bisher erledigten Jobs des gleichen JobTypes gebildet. Die so ermittelte Laufzeit wird klassifiziert und mit dem gespeicherten Leistungsindex des anfragenden Agenten verglichen. Dieser Index wird aus der Rechenleistung des Clients und seiner Zuverlässigkeit gebildet: Schnelle oder zuverlässige Rechner haben einen höheren Index als langsamere oder unzuverlässigere (zum Leistungsindex des Agenten siehe Abschnitt 3.2). Der Scheduling-Vorgang wird von drei Parametern $u,l,h, \in \{0,\ldots,100\}$ gesteuert und spielt sich wie folgt ab:

1) Es werden die beiden JobTypes bestimmt, welche die höchste und die niedrigste Anzahl an gerade arbeitenden Jobs besitzen. Sollte der kleinere der beiden Werte weniger als $u\,\%$ des größeren entsprechen, wird der erste Job desjenigen Typs ausgeliefert, von dem aktuell am wenigsten Jobs ausgeführt werden. Es wird also derjenige Benutzer bevorzugt, von dem gerade die wenigsten Jobs bearbeitet werden. Dies sorgt für ein gleichmäßiges Verteilen der Jobs unter den einzelnen Benutzern.
2) Befinden sich jedoch alle JobTypes innerhalb des von u festgelegten Bereiches, so wird folgendes Verfahren angewendet:
    a) Höchste Erledigt-Rate größer als h : Job dieses Typs ausliefern.
    b) Niedrigste Erledigt-Rate kleiner als l : Job dieses Typs ausliefern.
    c) Sonst: Job, dessen Typ am besten zur Leistungsklasse des anfragenden Agenten passt ausliefern.

Dieses Vorgehen sorgt dafür, dass sowohl neue Jobtypen als auch fast komplett erledigte Jobtypen beschleunigt werden und dass die Jobs entsprechend der Leistung der Agenten verteilt werden. Praxistests und Simulationen haben gezeigt, dass die Werte $u = 90$, $l = 10$ und $h = 90$ einen guten Kompromiss zwischen fairer und leistungsgerechter Verteilung darstellen. Die Zeitkomplexität des Verfahrens verhält sich wie folgt: Sei n die Anzahl der insgesamt vorhandenen Jobs und m die Anzahl der zur Agenten-

Anfrage passenden Job-Kandidaten. Dann gilt $m \leq n$. Die Jobbdatenbasis wird einmal komplett und einmal bis zum Job mit der gewählten ID durchlaufen, diese stehe an Position k, wobei $k \leq n$ gilt. Die Liste der Kandidaten wird ebenfalls einmal komplett durchlaufen. Das Schedulingverfahren hat folglich eine asymptotische Laufzeit von $O(n+k+m) = O(3n) = O(n)$, verhält sich somit linear zur Anzahl der vorhandenen Jobs.

3.2 Agent

Beim Agenten handelt es sich um den Teil des Systems, der auf den Clientrechnern läuft und für die Ausführung der Rechenjobs verantwortlich ist. Neben den im vorigen Abschnitt erläuterten Hauptaufgaben Job-Anfragen, Job-Download, Job-Start und Ergebnis-Upload kommen einige weitere Fähigkeiten hinzu:

- Beim Start des Agenten wird einmalig die Leistung des Systems gemessen und zum Server geschickt. Diese Daten werden beim Scheduling verwendet (s. Abschnitt 3.1).
- Um schmalbandige Netzzugänge möglichst wenig zu belasten wurde ein Dateicache implementiert, der verhindert, dass bereits heruntergeladene Dateien nochmals übertragen werden. Um Dateigleichheit (nahezu) sicherzustellen, tauschen Agent und Server MD5-Hashwerte der fraglichen Dateien aus.
- Die Rechenjobs werden auf dem jeweiligen System mit der niedrigsten Prozesspriorität gestartet um den Nutzer des Systems bei seiner Arbeit nicht zu stören[3]. Dabei werden im Hintergrund während der Ausführung permanent die Systemlast und der verfügbare freie Hauptspeicher gemessen. Ist die Last zu hoch oder sollte der freie Speicher zur Neige gehen, wird der Job abgebrochen.
- Der Agent kann auf dem Client-System mit eingeschränkten Benutzerrechten betrieben werden, er benötigt lediglich ein einziges Verzeichnis mit Schreibrechten 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.
- Sollte der Agent beim regelmäßigen Statusaustausch mit dem Server das Signal erhalten den Prozess abzubrechen, wird dies ebenfalls getan. Das geschieht z.B. dann, wenn der Entwickler auf dem Server einen Job löscht oder blockiert. In diesem Falle ist es nicht sinnvoll, den Job weiterhin auszuführen.
- Zwischen der Ausführung zweier Rechenjobs besorgt sich der Agent beim Server neue Anweisungen über den Type und die Plattform des nächsten Jobs. So kann man einen Agenten auf die Jobs eines bestimmten Benutzers festlegen und einem Benutzer mehr Rechenleistung zur Verfügung stellen als der Scheduler im Server dies tun würde.

---

[3] 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 nichts.

3.3 Parallelisierung und Management-GUI

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 $E = \{e_0, \ldots, e_n\}$,
- eine Befehlszeile C (oder ein Skript) das die Eingabedateien verarbeitet, und
- eine Menge von Ausgabedateien $A = \{a_0, \ldots a_m\}$, die durch die Befehlszeile oder das Skript erzeugt werden.

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

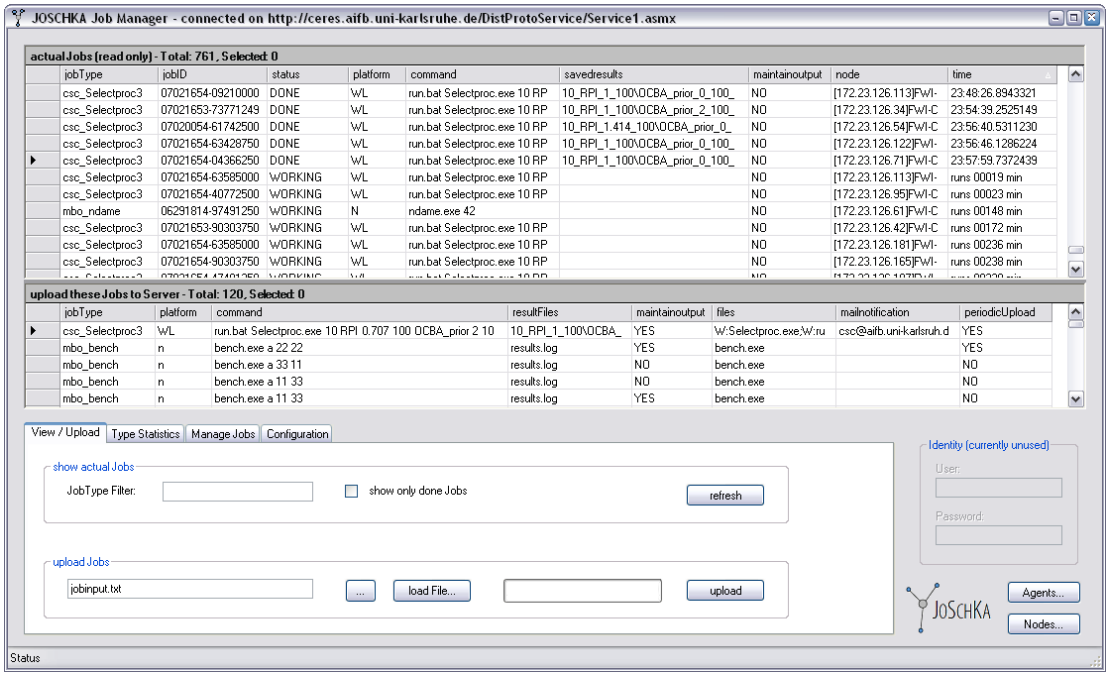

Abbildung 3: Jobmanagement-GUI

Abbildung 3 zeigt das Programm, mit dem der Entwickler seine Rechenjobs auf dem Server verfügbar machen kann. Nachdem über die Upload-Schnittstelle die Eingabedateien auf den Server kopiert wurden, muss in diesem GUI die untere Tabelle ausgefüllt werden, um die Daten per SOAP zum Server zu schicken, wo sie dann für die Agenten bereitgestellt werden. Die obere Tabelle zeigt den aktuellen Datenbestand an Jobs, die auf dem Server bereitliegen. Die Upload-Tabelle kann auch durch den Import einer Textdatei gefüllt werden. Diese Datei kann vom Entwickler z.B. skriptgesteuert selbst erzeugt werden.

4 Praxiserfahrungen

Der Einsatz im CIP-Pool[4] der Fakultät für Wirtschaftswissenschaften der Universität Karlsruhe (TH) mit handelsüblichen Windows- und Linux-PCs hat gezeigt, dass das beschriebene System einsatzfähig ist und einwandfrei funktioniert. Es stehen mehr als 100 Rechner der 3 GHz-Klasse zur Verfügung, die im Laufe von ca. 6 Monaten etwa 130.000 Rechenjobs aus dem Bereich der naturanalogen Optimierungsverfahren berechneten[5]. Die damit für wissenschaftliche Zwecke nutzbare Rechenleistung ist enorm. Die an den Rechnern interaktiv arbeitenden Studierenden bemerkten von alledem nichts. Ebenso können problemlos Büro- oder externe Privatrechner nach Bedarf mitrechnen, auf denen lediglich das Agentenprogramm laufen muss. Weitere Voraussetzungen sind nicht nötig. Das System eignet sich demnach sowohl für den Einsatz in homogenen Pool- bzw. Cluster-Umgebungen als auch für den Einsatz im sogenannten „Public-Ressource-Computing".

Literatur

[1] I. Foster, C. Kesselmann, „The Grid – Blueprint for a New Computing Infrastructure", Morgan Kaufman Publ. 1999

[2] Berkeley Open Infrastructure for Network Computing, http://boinc.berkeley.edu/

[3] zetaGrid, http://www.zetagrid.net/

[4] Portable Batch System PBSpro, http://www.pbspro.com/

[5] Cactus Code, http://www.cactuscode.org

[6] Condor High Throughput Computing, http://www.cs.wisc.edu/condor/

[7] GridKA im Forschungszentrum Karlsruhe, http://www.gridka.de

[8] The Globus Toolkit, http://www.globus.org

[9] Tuple-Spaces, http://c2.com/cgi/wiki?TupleSpace

[10] SOAP: Simple Object Access Protocol, http://www.w3.org/TR/soap/

[11] Microsoft .NET Framework, http://www.microsoft.com/net/

[12] Mono-Runtime, http://www.mono-project.com/

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

[5] Bei den Messläufen handelt es sich meist um zufallsgesteuerte Optimierungsverfahren, die oft hunderte oder tausende Male auf dem gleichen Problem simuliert werden, um sinnvolle Aussagen über ihre Leistungsfähigkeit und die ideale Parametrisierung machen zu können und sich deshalb hervorragend parallelisieren lassen. Getestet wurden Ameisenalgorithmen und evolutionäre Optimierungsverfahren.