DISSERTATION

A Distributed Computing Environment for Material Sciences

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Naturwissenschaften

unter der Leitung von
Univ.Prof. Dr.phil. Karlheinz Schwarz
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Blaha
E165 – Institut für Materialchemie

eingereicht an der Technischen Universität Wien
Fakultät für Technische Chemie

von

Dipl.Ing. Johannes M. Schweifer
Matrikelnummer: 9525661
Aspangstr 51/27, 1030 Wien

Wien, am 12. Dezember 2006

Kurzfassung:

WIEN2k ist eine materialwissenschaftliche Anwendung zur Berechnung der Elektronenstruktur von Festkörpern und basiert auf der Dichtefunktionaltheorie. Der Zeitaufwand solcher Rechnungen liegt im Bereich von Stunden bis Tagen und kann durch Parallelisierung verkürzt werden. Um möglichst viel Rechenleistung nützen zu können sollten alle verfügbaren Computer benutzt werden können, jedoch hat dies den Nachteil, daß der Anwender mit unterschiedlichen Betriebssystemen und Queuing Systemen arbeiten muß. Damit eine solche , zunehmend komplexe, verteilte Rechenumgebung auf einfache Weise genützt werden kann, sind Programme notwendig, die einen heterogenen Ressourcen Pool in eine homogene Umgebung - eine sogenannte Grid Umgebung - verwandeln. Die Schnittstelle zwischen der Hardware und den Anwendungen wird "Middleware" genannt. Leider stellt der Großteil der bereits verfügbaren Middleware hohe Anforderungen an Anwender, Entwickler und Systemadministratoren, weshalb im Rahmen dieser Dissertation nach einer neuen Lösung gesucht wurde, die nur die wichtigste Funktionalität einer solchen Middleware implementiert, wie beispielsweise die automatische Auswahl von Computern, Datenübertragung, die Durchführung einer Rechnung und deren Überwachung, jedoch ohne besondere Rechte auf dem Zielsystem oder grundlegendes Expertenwissen zu erfordern. Zu diesem Zweck wurde **W2GRID** entwickelt, eine auf Perl basierende Middleware mit minimalen Systemanforderungen. Die Neuheit dieser Entwicklung besteht darin, daß alle kritischen und von der Architektur abhängigen Aspekte von der Anwendung getrennt werden, wobei es aber gleichzeitig nicht notwendig ist, den Code derselben zu verändern. Statt dessen wird die Anwendung nur mit geeigneten Interface-Skripten gesteuert (Anwendungs-plugins). **W2GRID** ist kein monolithischer Code sondern eine Sammlung von voneinander unabhängigen Komponenten (plugins), die bei der Installation in passender Weise kombiniert werden. Somit kann man die Unterstützung für andere Betriebs- und Queuing-Systeme durch die Entwicklung entsprechender Plugins erweitern. Das Interface zwischen **W2GRID** und der Anwendung bleibt aber davon unabhängig. Die Middleware eignet sich sehr gut, um Programme innerhalb einer einzigen Rechnerdomäne auszuführen, ist aber nicht in der Lage über Domänengrenzen hinweg zu parallelisieren, denn eine solche Funktionalität ist nicht mit den Interface-Skripten zu bewerkstelligen. Im Gegensatz zu vielen anderen Middleware Produkten, die bereits eine bestimmte Infrastruktur und etliche Tools und Dienste voraussetzen, um auf diesen aufbauen zu können, beschränkt sich **W2GRID** als "Standalone-Lösung" nur auf die wichtigsten Aspekte von verteiltem Rechnen. Daher ist der Anwender nicht gezwungen zusätzliche Software zu installieren, denn **W2GRID** benötigt nur jene Tools und Bibliotheken, die auf den üblichen Unix/Linux basierten Rechnern standardmäßig vorhanden sind. Die vorliegende Arbeit beschreibt die grundlegenden Konzepte der Middleware und die Entwicklung des Anwendungs-plugins für **WIEN2k**, das es dem Anwender ermöglicht sich auf seine wissenschaftliche Arbeit zu konzentrieren und die Rechenarbeit einem automatisierten Prozeß zu überlassen. Das **WIEN2k** plugin bewertet die Rechenaufgabe in Hinblick auf die erforderlichen Ressourcen, fordert die momentane Auslastung der verwendbaren Rechner an und wählt auf dieser Basis das am besten geeignete Computersystem aus. Die Input Dateien werden auf den Zielrechner kopiert, die Rechnung gestartet und überwacht, wobei die Output Dateien in regelmäßigen Abständen auf dem lokalen Rechner auf den neuesten Stand gebracht werden. Manche davon müssen nicht als ganzes kopiert werden, daher reicht es, das jeweils aktuellste Fragment an die bereits existierende Datei anzuhängen. Während der Rechnung wird die Lastverteilung dynamisch an eine sich verändernde Auslastung angepaßt, wodurch die vorhandenen Ressourcen effektiver ausgenutzt werden können. Das Plugin wurde entwickelt, um die mittlerweile 1000 wissenschaftliche Gruppen und Firmen umfassende **WIEN2k** Gemeinde bei der Verwendung von verteilten heterogenen Rechenumgebungen zu unterstützen. Schließlich wurde gezeigt, daß auch andere Anwendungen mit ähnlichen Anforderungen **W2GRID** nutzen können indem ein entsprechendes Plugin entwickelt wird.

Abstract:

WIEN2k is a material science application, which uses Density Functional Theory to calculate the electronic structure of solids. Such calculations can take many hours up to several days, therefore parallelisation is used to speed up the computations. In order to harvest as much computing power as possible, it is desirable to employ all hardware available to a scientist. This, however, comes with the unfavourable disadvantage, that the user has to cope with different operating systems and methods of job-submission. To cope with such an increasingly complex and distributed environment, certain tools are necessary, which turn a heterogeneous pool of computing resources into a homogeneous one, forming a so called “Grid environment” and provide an interface layer in between the application and the hardware, termed middleware. The majority of existing middleware solutions come with an unfavourable overhead of requirements for users, developers and system administrators. Therefore the motivation of this work was to find novel solutions for the most basic features an application needs in a distributed computing environment, namely automated resource selection, filetransfer, job-submission and its monitoring, without strict requirements in terms of permissions and experience. For this purpose **W2GRID**, a lightweight Perl-based middleware, has been developed. It represents a novel approach to address the increasing heterogeneity of distributed resources by separating all critical architecture dependent aspects of scientific computing from the application, but obviates the necessity to modify the source code of the application, instead the latter is wrapped with proper interface scripts (application plugins). **W2GRID** does not come as a monolithic code with limited portability but instead consists of a collection of independent components (plugins), which are combined during the installation in such a way, that support for new job-submission schemes or new operating systems can be added by developing new plugins, but the interface in between the application and **W2GRID** remains independent of both. The middleware is well suited to run programs within a single domain having a shared filesystem, whereas it is not possible to parallelise programs across several computing sites, since parallelism cannot be added with **W2GRID** but must already be provided by the application. In contrast to numerous other middleware schemes, which are built on top of an already existing infrastructure or certain high-level tools, **W2GRID** restricts itself to the essential parts of distributed computing and provides a standalone solution, which can interact with a variety of infrastructures by the means of proper plugins. The user does not have to install additional software, since **W2GRID** is free of third party dependencies. It does not need tools or libraries other than those, which can be expected on any Unix/Linux based system. The presented work covers the design and implementation of the middleware as well as the development of the application plugin, which allows the user to focus on the material science problem and to leave all computational tasks to an automated scheme. The **WIEN2k** plugin analyses the submitted task with respect to its requirements, collects immediate status-informations of all suitable computing resources and selects the best one for this task. It transfers the input files to the remote host, starts the calculation, monitors it and continously updates the local output files, some of which are not copied repeatedly as a whole, but instead are just appended by the most recent chunk of data. During the execution, the load-distribution can be adjusted dynamically leading to an efficient and economic use of all available computing resources. The plugin is designed to promote the use of distributed computing in the growing **`WIEN2k`** user community, which consists already of more than 1000 academic workgroups and companies. Finally it was shown that **`W2GRID`** can also be used for other scientific applications with similar demands by writing the corresponding application plugin.

Acknowledgements

I want to express my gratitude to my two supervisors Karlheinz Schwarz and Peter Blaha, who gave me the chance to pursue this interesting topic, which proved to be exactly the kind of interdisciplinary work I had been searching for. The knowledge I gained in their workgroup was a big leap for me, and I am grateful, that they supported the development of **`W2GRID`** up to its present stage, as it has originally not been supposed to serve also other applications.

Sincere thanks are due to Prof. Armin Scrinzi of the Photonics Institute for the many fruitful discussions we enjoyed at the occasion of numerous AURORA meetings and also for his willingness to employ **`W2GRID`** on their local computing infrastructure. I also want to thank Christopher Ede for his effort to do the alpha and beta testing of **`W2GRID`**. It was mainly due to his feedback, that numerous bugs of the middleware when used in the Bourn-Shell could be detected and fixed. It was also him, who contributed the application plugin for **`MCTDHF`**.

Furthermore I am delighted to thank my colleagues (in alphabetical order) Clemens Först, Thomas Gallauner, Robert Laskowsky, Andreas Mattern, Günther Schmidt, Bernd Sonalkar and Christian Spiel for the atmosphere of friendship during these last four years. I am grateful that I could enjoy their expertise in numerous scientific and non-scientific fields. Basically there was always a person, who knew a solution for the sometimes difficult problems with LaTeX, C or the shell scripting languages.

I will not forget to express my gratitude to my parents, since my studies would not have been possible without their help and encouragement.

My ultimate thanks, however, belong to my partner Michaela, as she provides me with continuous support. Without her, the days I experienced during writing up this thesis would have been much tougher to endure.

Contents

Chapter 1

Introduction

Chemistry and physics increasingly employ computational methods in order to study and understand matter. In these evolving disciplines, which became known as 'computational material sciences' the computer itself is the primary instrument of research, distinct from its use for the storage and display of results obtained from experiments. While scientists and application developers can profit on the one hand from the formidable hardware progress in recent years, they already encounter problems due to the rising number of mostly different computing architectures on the other hand. It is therefore aimed to reduce the administrative overhead, which is inevitably related with heterogeneous computing pools.

The motivation of this thesis was to investigate the impact of distributed resources on **`WIEN2k`**, a quantum mechanical electronic structure code for solids. It aims to find a solution, which allows scientists to focus on their material science problem while leaving all the computational details of the (often time-intensive) calculations to an automated scheme that runs in the background.

Soon after this project was launched, a survey on existing infrastructures serving a similar purpose made apparent that the task could not be solved by the use of any of them without either having to make vast changes to the **`WIEN2k`**-code[1] or imposing a number of unfavourable requirements that limit its use for many scientific workgroups. Therefore it was decided to pursue a proprietary development, which was named **`W2GRID`**[2] with respect to its initial purpose. In a later stage of development the concept was generalised to serve also the demands of other scientific applications such as **`MCTDHF`**.

Since both applications are based on quantum mechanics, the thesis will introduce the reader into its basics as this is required to understand the scientific background of the applications. Furthermore it is considered very helpful to explain the underlying principle of material sci-

[1]hence producing a very architecture-dependent solution, which is barely portable.

[2]WIEN-to-grid

ence within the scope of this document as it eases the understanding of the demands of such applications and the problems for scientists and developers, which arise from the work in a heterogeneous distributed environment. A detailed description thereof is given next, followed by a survey of the commonly used infrastructures, which are offered as a generalised but heavy-weight solution. The introduction is concluded with a list of reasons, why these solutions are far from optimal for certain applications such as **`WIEN2k`** or **`MCTDHF`** and how **`W2GRID`** can help to improve the usage of scientific applications on a pool of remote user-accounts. Chapter 2 and 3 describe its design-concepts and the way how third party software can be integrated, which is demonstrated on the basis of **`WIEN2k`** in chapter 4. The proof that both, the underlying principle of architecture-independence as well as the presented implementation of the **`WIEN2k`**-specific solution works as desired, is finally given in chapter 5.
Besides its original purpose to find a portable solution for the **`WIEN2k`** community, **`W2GRID`** can be used to run almost any application on distributed resources provided that the proper interface is implemented. Therefore this thesis shall additionally serve as a guide for developers.

1.1 Computational Material Sciences

This field of research allows to predict physical properties of materials and to study their behaviour in certain environments before they are synthesised and analysed in the laboratory. Computational material sciences may be used as a guide to exclude less favourable reactions or unstable products, or to select more fruitful reaction-paths from the many possible ones. An additional benefit is the possibility to examine effects on a scale of space or time that is difficult to reach by present experimental techniques, thus giving an insight beyond that provided by experiments. The success of this field is based on quantum mechanics[3], which gave us the fundamentals to develop new modern materials (e.g. optic/magnetic storage devices, sensors, shape memory alloys, the laser or magnetic resonance imaging). The study of semiconductors for example led to the invention of the diode and the transistor, which are indispensable for modern electronics.
The following text is based on the literature [1, 2, 3, 4, 5, 6], lecture-notes and the Internet.

1.1.1 Quantum mechanics

Quantum mechanics is the underlying physical, mathematical and theoretical framework to explain, study and simulate matter. It evolved in the beginning of the twentieth century from the aim to understand matter on an atomic scale and below, when classical mechanics proved to

[3]The term ’quantum’ is Latin (’how much’) and refers to the property of waves being measurable in particle-like discrete packets of energy called ’quanta’.

be inadequate for this purpose. Experiments led to a theory of unity between subatomic particles and electromagnetic waves called **wave-particle duality**, in which particles and waves were neither one nor the other, but had certain properties of both, giving rise to a fierce search for new models to explain the experiments. Based on the results, Niels Bohr proposed a planetary model for the atom, with electrons orbiting a sun-like nucleus. To stabilise the system, which would otherwise collapse, he had to introduce a few fundamental postulates. Bohr's model agrees with the hydrogen atom but fails to explain any of the heavier elements and thus lacks an approach to chemical bonding. The next step by Erwin Schrödinger, who employed wavefunctions ψ to describe matter, was a resounding success and made quantum mechanics the pillar of modern physics.

1.1.1.1 The Schrödinger equation

On the basis of the wave-particle dualism it was concluded, that matter can be described as a superposition of numerous waves of different frequencies[4], amplifying in a certain region of space and destructively interfering elsewhere. In this approach all particles are described by a partial differential equation (1.1) of a wavefunction (ψ) in space (r) and time (t) where $V(r)$ is a given potential energy.

$$i\hbar\frac{\delta\psi(r,t)}{\delta t} = -\frac{\hbar^2}{2m}\frac{\delta^2\psi(r,t)}{\delta r^2} + V(r)\psi(r,t) \tag{1.1}$$

This is the fundamental non-relativistic equation of quantum mechanics and was first proposed by Schrödinger. It is very often referred to as the corresponding counterpart of Newton's second law and simplified into equation 1.2.

$$i\hbar\frac{\delta\psi(r,t)}{\delta t} = -\mathbf{H}\psi(r,t) \tag{1.2}$$

where $\mathbf{H}$ is the Hamilton operator[5]. In a stationary state, the total energy of the system is constant. Separating the variables and removing the time-dependence leads to the well known time-independent Schrödinger equation 1.3

$$\mathbf{H}\psi = E\psi \tag{1.3}$$

[4]De Broglie related the momentum of a particle to its wavelength $p = h/\lambda$, which allowed to calculate the quantum wavelength of electrons by the knowledge of their momentum.

[5]The Hamilton operator is the counterpart of the Hamilton-function in classical mechanics. An introduction to operators is given in [2]

where E is a scalar eigenvalue and ψ is an eigenfunction, which must be unique and continuous, such that the integral $\int |\psi|^2 \, \mathrm{d}\tau = A$ is finite[6] so that ψ can be normalised.
Trajectories are not known in quantum mechanics due to Heisenberg's uncertainty principle, saying that the position and the momentum of a particle cannot be determined exactly at the same time ($\Delta x \Delta p \geqq \frac{\hbar}{2}$). Instead, the position of a particle is described by a probability of finding it within a certain volume of space, expressed as the product of the wavefunction with its complex conjugated counterpart $\psi\psi^*$. This probability of finding the particle in a finite volume of space, requires that the wavefunction is normalised by $\int |\psi|^2 \, \mathrm{d}\tau = 1$
In contrast to the Bohr model, the Schrödinger equation allows the electrons to occupy three-dimensional regions of space, hence their description requires three coordinates. Solving Schrödinger's equation for the hydrogen atom, which is colourfully demonstrated in [1], yields three quantum numbers (QN), that are well known to chemists:

- principal QN (**n**): defines the sizes of the orbitals
- angular QN (**l**): defines the shape (spherical, polar, cloverleaf,..)
- magnetic QN (**m**): defines the orientation in space

The intriguing fundamental postulate of quantum mechanics is that a wavefunction ψ exists for any (chemical) system. Once it is obtained, one can go from an appropriate operator (function) acting upon ψ to any desired observable by calculating the 'expectation value'[7].

$$< \alpha >= \frac{\int \psi^* \alpha \psi \mathrm{d}\tau}{\int \psi^* \psi \mathrm{d}\tau} \tag{1.4}$$

This is expressed by equation 1.4, where α is an operator, ψ is an eigenfunction to $\mathbf{H}$ but not necessarily to α. If the distribution is sharp ($\alpha\psi = a\psi$), the expectation value is the eigenvalue ($< \alpha >= a$).

1.1.1.2 The electron many-body problem

Unfortunately the Schrödinger equation, being the ultimate key to all properties of interest, is unsolvable in all practical cases except for the most simple system, namely the hydrogen atom. To account for the interactions of electrons and nuclei the Hamilton-operator can be written according to equation 1.5, where i and j run over electrons, k and l run over nuclei, m_e is the mass of the electron, m_k is the mass of the nucleus k, ∇^2 is the Laplacian operator

[6] $\mathrm{d}\tau = \mathrm{d}x\mathrm{d}y\mathrm{d}z$

[7] If one measures this quantity in a series of experiments, the expectation value is the average value of all the results. This average can also be calculated.

($\triangle$), e is the charge of an electron, Z_k is the atomic number of atom k, and r_{ab} is the distance between two particles a and b.

$$\mathbf{H} = -\sum_i \frac{\hbar^2}{2m_e}\nabla_i^2 - \sum_k \frac{\hbar^2}{2m_k}\nabla_k^2 - \sum_i\sum_k \frac{e^2 Z_k}{r_{ik}} + \sum_{i<j}\frac{e^2}{r_{ij}} + \sum_{k<l}\frac{e^2 Z_k Z_l}{r_{kl}} \tag{1.5}$$

In Cartesian coordinates, the Laplacian has the form of equation 1.6.

$$\nabla_i^2 = \frac{\partial^2}{\partial x_i^2} + \frac{\partial^2}{\partial y_i^2} + \frac{\partial^2}{\partial z_i^2} \tag{1.6}$$

The Hamiltonian $\mathbf{H}$ contains contributions from the kinetic (the first two terms) and potential energy (the last three terms) of all particles (electrons and nuclei) where the latter appear exactly as in classical mechanics. Since an analytic solution cannot be obtained, the problem must be simplified by approximations and constraints. The **Born Oppenheimer Approximation** is based on the fact, that the nuclei are much heavier and thus move much slower than the electrons, hence their motions can be assumed to be independent. This allows to eliminate the kinetic energy of the nuclei, and the nuclei-nuclei interaction becomes a constant for a given geometry. The new equation for the electrons is written as:

$$(H_{el} + V_N)\psi_{el} = E_{el}\psi_{el} \tag{1.7}$$

Yet the remaining treatment of the electronic structure is still challenging. According to equation 1.4 every observable can be calculated from the wavefunction. Since there are many possible solutions but only a single 'exact' one, which cannot be obtained directly, one is helped by the **variational principle**. It governs, that a calculated observable (expectation value of an operator) for any approximate wavefunction is greater or equal the exact value. Hence a systematic variation of this trial function allows to minimise the result, which finally yields a wavefunction close to the exact ψ.

Many approaches have been developed to explain chemical bonding and to find useful approximations. The interested reader is referred to [2, 3] for the LCAO model, the Hückel method and related theories. In the following sections two often used schemes are outlined, namely the 'Hartree-Fock' and 'DFT' method. In order to simplify the equations atomic units will be used, which are listed in table 1.1.

symbol	***Quantity***	***Value in a.u***	***Value in SI units***
m_e	mass of an electron	1	$9.110 \cdot 10^{-31}$kg
e	charge of an electron	1	$1.603 \cdot 10^{-19}$C
$\hbar$	momentum ($\frac{h}{2\pi}$)	1	$1.055 \cdot 10^{-34}$Js
a_0	Bohr radius (distance)	1	$5.292 \cdot 10^{-11}$m
E_H	Hartree (energy)	1	$4.360 \cdot 10^{-18}$J

Table 1.1: Atomic units

1.1.1.3 Hartree Fock

The fundamental assumption of the HF theory is to have independent electrons, which obey the Pauli principle. Each electron sees all the others as an average field. If the Hamiltonian can be written as a sum of one-electron operators with the kinetic energy, the nuclear attraction and the Coulomb interaction with the other electrons, the operator is separable and may be expressed as

$$\mathbf{H} = \sum_{i=1}^{N} h_i = -\frac{1}{2}\sum_{i=1}^{N} \nabla_i^2 - \sum_{k=1}^{M} \frac{Z_k}{r_{ik}} + \sum_{j \neq i} \int \frac{\rho_j}{r_{ij}} \mathrm{d}r \tag{1.8}$$

where N is the total number of electrons, h_i is the one-electron Hamiltonian and ρ_j is the charge density associated with electron j. Eigenfunctions of the one-electron Hamiltonian must satisfy the one-electron Schrödinger equation, therefore the many-electron eigenfunction ψ can be constructed as products of the one-electron eigenfunctions ϕ_i. Equation 1.9 is called a 'product ansatz' and ψ_{HP} a Hartree product-wavefunction.

$$\psi_{HP} = \phi_1 \phi_2 \cdots \phi_N \tag{1.9}$$

The problem is, that calculating the wavefunction ϕ_i requires the knowledge of the other densities $\rho_j = |\rho_j|^2$, which in turn depend on the wavefunction again. To solve this problem, Hartree proposed an iterative 'self-consistent field' (SCF) method. In the first step, one *guesses* the wavefunctions ϕ_i for all of the occupied orbitals and uses them to construct the necessary one-electron operators h_i. The solution of each differential equation provides a new set of functions ϕ, different from the initial guess. This procedure is continued iteratively until the difference between the old and the new functions converges within a certain threshold criterion.
To account for the electron-spin and the Pauli principle, namely that the many-electron wavefunction must be antisymmetric, Fock extended the Hartree product by a Slater determinant of

the following form:

$$\psi_{SD} = \frac{1}{\sqrt{N!}} \begin{vmatrix} \chi_1(1) & \chi_2(1) & \cdots & \chi_N(1) \\ \chi_1(2) & \chi_2(2) & \cdots & \chi_N(2) \\ \vdots & \vdots & \ddots & \vdots \\ \chi_1(N) & \chi_2(N) & \cdots & \chi_N(N) \end{vmatrix} \tag{1.10}$$

where N is again the total number of electrons and $\chi_i(n)$ are 'spin orbitals', a product of the spacial orbital ϕ_i and an electron-spin eigenfunction (for spin-up or spin-down).
HF theory treats exchange exactly but ignores correlation, and thus cannot realistically be used to represent real systems with chemical accuracy, for example if one is interested in quantities such as the heat of formation.

1.1.1.4 DFT

Density Functional Theory (DFT) is one of the most popular and successful quantum mechanical approaches to matter. Nowadays it is routinely applied for calculating, e.g., the binding energy of molecules in chemistry and the band structure of solids in physics. The advantage of DFT is that it is not necessary to calculate the complicated N-electron wavefunction $\psi(x_1, x_2, \cdots, x_N)$, but instead one can restrict oneself to calculating the much simpler electron density ($\rho(r)$), a fact that was first proven by Hohenberg and Kohn. The practical scheme that they devised is based on a hypothetical reference system of noninteracting electrons, chosen in such a way that the density of this system is identical to the exact density of the real physical system under consideration. The Hohenberg-Kohn theorem is exact and says: *"The total energy of an interacting inhomogeneous electron gas in the presence of an external potential $V_{ext}(r)$ is a functional of the density $\rho(r)$".*

$$E_{tot}(\rho) = T_s(\rho) + E_{ee}(\rho) + E_{Ne}(\rho) + E_{xc}(\rho) + E_{NN} \tag{1.11}$$

Where T_s is the kinetic energy term for the non-interacting particles, E_{ee} accounts for electron-electron and E_{Ne} for the nucleus-electron interactions, E_{NN} represents the Coulomb interaction between the nuclei and finally E_{xc} adds the exchange-correlation energy, which contains the difference between the energy of the system calculated by the other four contributions and the 'exact' energy. The many-body problem of interacting electrons and nuclei is mapped to a series of one-electron Schrödinger-like (so called Kohn-Sham) [7] partial differential equations, that lead to the same density as the real system.

$$[-\frac{1}{2}\nabla_i^2 + V_i]\Phi_i = E_i\Phi_i. \tag{1.12}$$

$$\rho(r) = \sum_{i}^{N} |\phi_i|^2 \tag{1.13}$$

While their theorems show, that it is possible to use the ground state density to calculate properties of the system, there is no way provided to find the ground state density, because the approach actually does not solve the equation, it only shifts the so far insoluble parts to E_{xc}, whereas all the other expressions can be calculated very accurately. Finally E_{xc} is obtained from different approximation techniques (e.g. LDA). The interested reader is referred to the literature [8, 9]. The partial differential equations 1.12 correspond to an eigenvalue problem with the eigenfunctions Φ_i and the eigenvalues E_i, which can only be solved iteratively, since the potential requires the knowledge of the density, but the density is computed from the sum $\Phi_i^* \Phi_i$ over all occupied electronic states i, which need the potential for obtaining the corresponding one-electron Kohn-Sham orbitals Φ_i. The procedure is already known from Hartree-Fock and is illustrated in Fig.1.1. The electronic, magnetic, mechanical or optical

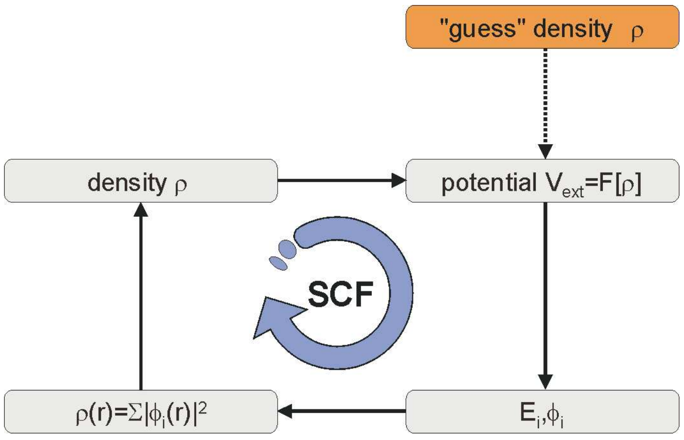


Figure 1.1: The iterative SCF-scheme

properties of many systems (molecules and solids) can all be studied with DFT.

1.1.2 Material Science applications

Simulations in material science can be characterised by the following three types, 'ab initio', 'semi-empirical' and 'molecular mechanics' calculations.

- **ab initio**: Latin for "from scratch" stands for "first principles" methods. Molecular or solid structures can be calculated based on quantum mechanics using nothing but the Schrödinger equation, the values of the fundamental constants and the atomic numbers of the constituent atoms. The described methods of DFT and HF belong to this category.

- **semi-empirical**: Such techniques simplify the calculations by the use of approximations (e.g. for integrals, which rely on empirical data) to adjust results to experimental values.
- **molecular mechanics**: Uses classical physics and empirical or semi-empirical (predetermined) force fields (based on balls and springs) to explain and interpret the behaviour of atoms and molecules. No information on the electronic structure can be obtained from such a classical treatment. However it can model the atomic structure of very large size.

Popular applications besides **`WIEN2k`** and **`MCTDHF`**, which are discussed in the following sections, are among many others ***Gaussian***, ***ABINIT***, ***GROMACS***, ***SIESTA***, ***AMBER*** or ***VASP***.

1.1.3 WIEN2k

`WIEN2k` is a material science application, which uses Density Functional Theory [8] to calculate the electronic structure of a given solid [9, 10]. It was developed in the group during the last 25 years.

1.1.3.1 Scientific background

`WIEN2k` calculates the electronic structure of crystalline solids, which are based on a certain basic structural unit infinitely repeated in all three dimensions. This reduces the problem with infinite boundaries to a calculation of a 'small' or at least finite number of atoms within the unit cell. It is assumed, that the material can be described as an ideal single-crystal having a well defined stoichiometry and a fixed translational symmetry, which allows the calculation to be done in reciprocal space, whose base-vectors are orthogonal to the base-vectors in real space. The reciprocal lattice basis-vectors span a vector-space that is commonly referred to as reciprocal space, or often in the context of quantum mechanics, k-space. One of the points in the reciprocal lattice is then designated to be the origin. The unit cell in reciprocal space is obtained by drawing lines from this origin to all nearby (closest) lattice points. At the midpoint of each line between the lattice points a perpendicular plane is drawn. The resulting cell is called the first **Brillouin-zone**. As a result of the Bloch theorem, the general solution of the wavefunctions are Bloch functions that can be expressed by plane-waves (sinusoidal functions). What distinguishes the linearised augmented plane wave (LAPW) method from others is the choice of the basis set. The LAPW basis is constructed to be particularly accurate and efficient for the solution of the all-electron ab initio electronic structure problem, where solutions are rapidly varying and atomic-like (like isolated-atom solutions) near the nuclei but are smoothly varying throughout the rest of the cell. Close to the nucleus, the oscillations of the wavefunctions are significantly more vigorous than in the distance. To describe these oscillations properly, a large number of plane-waves would be needed, hence the idea of LAPW is to reduce the number of basis functions by separating the unit cell of a crystal into two types of regions: The atomic spheres and the so-called interstitial region. The radius of the spheres is called the muffin-tin radius, which should be chosen as big as possible as long as the radii of different atoms do not overlap. The LAPW basis functions are then constructed by connecting plane-waves in the interstitial region to linear combinations of atomic-like functions inside the spheres. This ansatz leads to a linear-eigenvalue problem, which is solved iteratively by the SCF-method described earlier.

1.1.3.2 Important features and some computable quantities

`WIEN2k` allows to calculate the total energy and the electron-density of a crystal, which may be used to compute many quantities, represented by a few examples:

- **Total Energy (ENE):** Allows to calculate the relative phase stability and the equilibrium lattice constants.
- **Density-of-states (DOS):** is a property that quantifies how closely packed energy levels are in some physical system and gives the number of states per unit volume in an energy interval.
- **Electric Field Gradient (EFG):** is an important structural property of a crystalline solid, which is defined at the nuclear site. The EFG is non-zero only if the charge surrounding the nucleus deviates from spherical symmetry and thus generates an inhomogeneous electric field at the position of the nucleus. It can be measured by nuclear quadrupole interactions using NMR or NQR.
- **Bandstructure:** describes energy bands of electrons. The band structure determines a material's electronic, optical, and a variety of other properties.

By calculating the forces (**FOR**) acting on each atom, a **geometry optimisation** is possible by moving the atoms according to the force-vectors to reach equilibrium, where all forces should vanish.

1.1.3.3 Workflows

`WIEN2k` consists of a set of independent Fortran programs[8] that are linked by C-Shell scripts[9], which directly represent the workflow of an SCF-calculation. Apart from the most frequently

[8] e.g. LAPW0, LAPW1, LAPW2, LCORE and MIXER
[9] Such an implementation is quite common in science [11] due to its flexibility

used example illustrated in Fig.1.2, half a dozen other workflows (e.g. runsp_lapw, runafm_lapw, runfsm_lapw) are available, which basically represent modifications or extensions of the one presented. The SCF cycle, however, remains the elemental principle. The left-hand side of

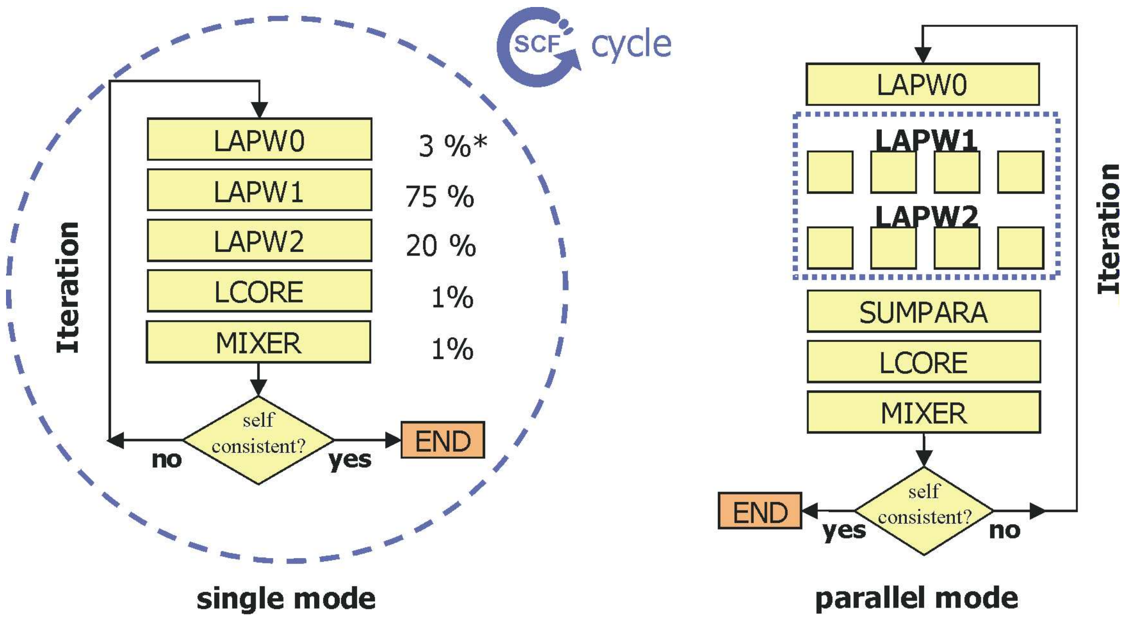


Figure 1.2: Sample workfbw of a self-consistent-fi eld cycle (SCF) of **`WIEN2k`**

Fig.1.2 represents a single mode run. Different executables are invoked sequentially, and the output of one process is usually the input of the next one. The computation time for the whole SCF calculation can range from a few seconds to several hours depending on the input. The average fraction of computation time for individual executables is given right to the boxes (*). In order to speed up the calculation, the most time-consuming processes (e.g. LAPW1, LAPW2) may be split into independent tasks (chunks) and run in parallel (right-hand side), if the number of k-points permits it. Otherwise the task has to be MPI-parallelised. For details see [12]. The SCF-cycle is the basic element of all kinds of tasks a scientist may perform by using **`WIEN2k`**, hence itself can be an element in larger workflows like phonon calculations or structure optimisations (Fig.1.3).

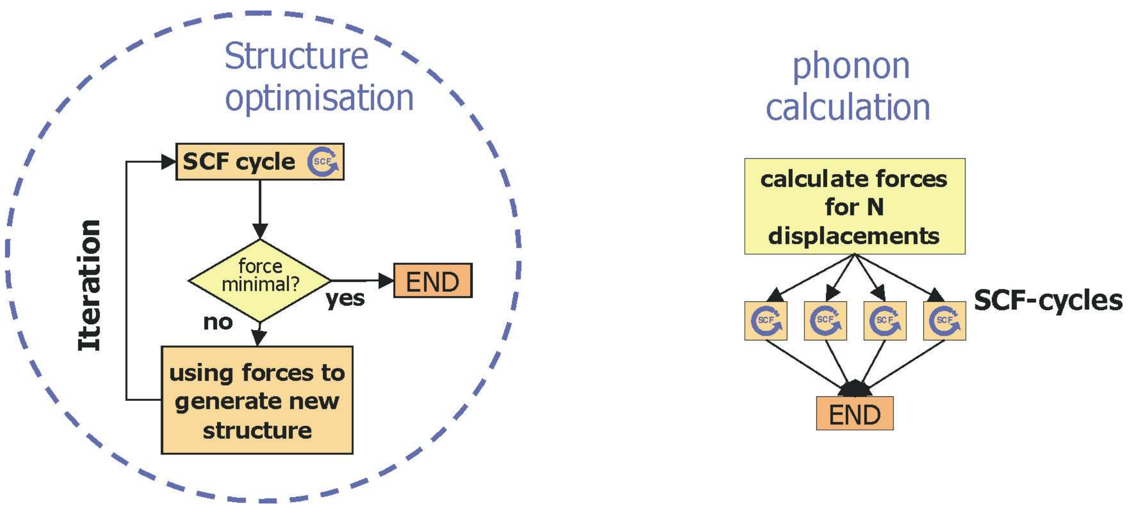


Figure 1.3: Example of two workfbws, which are built on top of the SCF-cycle (phonon calculation and structure optimisation)

1.1.3.4 Definition of a **`WIEN2k`**-CASE

A specific calculation is referred to as a 'CASE' (e.g. 'graphite'). It is usually characterised by the crystal structure and special settings defined in various input files for the particular calculation. In the naming convention of **`WIEN2k`**, this CASE-name is used for the directory name, which contains all data. Additionally all important input- and output files are named after the CASE, only the file extension will define their content and purpose (e.g. the file 'CASE.struct' contains the lattice, the atomic positions and the crystal symmetry) . A sample

```
user@localhost:/home/user/lapw/NaF> ll NaF.*
 32 -rw-rw-r--   31569 2006-10-31 03:53 NaF.clmcor
216 -rw-rw-r--  213880 2006-10-31 03:53 NaF.clmsum
200 -rw-rw-r--  197938 2006-10-31 03:53 NaF.clmval
  4 -rw-rw-r--     666 2006-10-31 03:53 NaF.dayfile
 16 -rw-rw-r--   12611 2006-10-31 03:53 NaF.energy
  4 -rw-rw-r--      70 2006-10-29 19:26 NaF.in0
  4 -rw-rw-r--     503 2006-10-29 19:26 NaF.in1
  4 -rw-rw-r--     291 2006-10-29 19:26 NaF.in2
  4 -rw-rw-r--     189 2006-10-29 19:26 NaF.inc
  4 -rw-rw-r--     170 2006-10-29 19:26 NaF.inm
  4 -rw-rw-r--     161 2006-10-29 19:26 NaF.inst
  8 -rw-rw-r--    5196 2006-10-29 19:26 NaF.kgen
  4 -rw-rw-r--     765 2006-10-29 19:26 NaF.klist
 32 -rw-rw-r--   28900 2006-10-31 03:53 NaF.scf
 20 -rw-r--r--   16701 2006-10-29 19:26 NaF.struct
168 -rw-rw-r--  166610 2006-10-31 03:51 NaF.vns
 32 -rw-rw-r--   31473 2006-10-31 03:51 NaF.vsp
```

Figure 1.4: A sample directory listing of a the CASE 'NaF' (truncated)

directory listing of a few CASE-files is shown in Fig.1.4, in which the name 'NaF' (sodium fluoride) is used for the files as well as for the name of the directory.

1.1.4 MCTDHF

The motion of several electrons in the presence of a strong field is a complex wave-packet problem that can only be treated by numerical methods. Straight-forward discretisations of the time-dependent Schroedinger equation lead to huge numerical systems already in the case of only two electrons, which is troublesome since even the simplest system investigated in attosecond physics (core-hole formation and Auger decay) involves the motion of three electrons. For such systems, the group of A. Scrinzi [13] pioneered the development of the Multi-Configuration Time-Dependent Hartree-Fock (MCTDHF) method [14, 15], which allowed for the first time three-dimensional ab initio calculations of molecules in strong laser pulses. It takes an intermediate position between a full solution of the time-dependent Schrödinger equation and the time-dependent Density Functional Theory.

1.1.4.1 Scientific background

In Hartree-Fock methods the full N-electron wavefunction $\psi q_1, q_2, \ldots, q_N$ is approximated in terms of products of single-electron orbitals ϕ_i, expressed as a single Slater determinant (or “configuration”) to account for the antisymmetry of the many-electron wavefunction ψ. The multiconfigurational Hartree-Fock ansatz consists of a linear combination of all Slater determinants that can be formed from N linearly independent orbitals. Its time-dependent version is particularly powerful for calculating the interaction of few-electron systems: By allowing the factor functions $\phi(x;t)$ to adjust with time, the wavefunction is expanded only on those parts of the space where it has significant amplitude.

1.1.4.2 Computational basics

The effort for the calculations grows as N^4. The code is MPI-parallelised [16] and shows nearly linear scaling of the runtime, which is in the range of hours on a parallel computer. A typical calculation consumes roughly 500 MB of memory.

1.2 Distributed Computing

1.2.1 Challenges for Scientifi c HPC applications

The computational effort for the simulation of material properties depends on the applied method and the problem size. While scientists are constantly improving their algorithms, the rising demands on the precision of the obtained results and the increased complexity of interesting modern materials, require extensive computing resources. We all know and accept the fact that experiments are often costly but we rarely realise, that computational studies can be expensive too. It is not only the expenses for the hardware but also the costs of operation, predominantly the power consumption of the processors on the one hand and the same amount of energy required for cooling. Consequently computational material science is an expensive scientific discipline, although its costs may be calculated differently.

The hardware, which is available for scientific computing today is different to the one, which has been in use less than a decade ago, since the large multi-processor supercomputing systems at the local university [17] have been replaced by clusters and arrays of idle desktop computers to be the dominant resources for researchers. The change is due to a matter of cost rather than technology [18]. Propelled by Moore's law [19] the increasing power of commodity PC's allow to build highly scalable clusters [20], which yield the same computing power at significantly lower cost than a multi-processor shared memory machine. As a consequence, application developers had to change their programming paradigms and focus on effective means of parallelisation and load distribution. This task is becoming even more challenging, since the number of nodes and CPUs on a cluster and even the number of cores per CPU are rising. To support the implementation of parallel tasks, application designers can rely on powerful libraries and toolkits such as the message-passing interface 'MPI' [21] (or the parallel-virtual-machine 'PVM' [22]). On the other hand present-days hardware is far more short-lived [23] than a decade ago. Commodity clusters are expected to yield an acceptable performance/cost ratio not longer than 3-6 years until they have to be replaced, hence the application must be flexible and portable and as hardware-independent as possible, otherwise it will require significant adaptations every couple of years.

1.2.2 Challenges for the Scientist

The inevitable impact on the researcher, who uses such a (parallel) application in a distributed environment, is a significantly increased effort. The control of an array of PC's in a commodity cluster is usually dedicated to a cluster management software [24] like e.g. PBS [25] or the Sun-Grid-Engine [26], which on the one hand simplifies the administration and the sharing of the hardware in a multi-user environment, but on the other hand uses its very unique syntax to negotiate computing resources. In most cases different clusters will be equipped with a different type or version of such a cluster management software[10], demanding quite some expertise from the user.

Furthermore parallelism was not the only hardware-driven change the scientific community had to face in the recent past. The Internet, which could in its infancy only provide minor data transfer rates, barely sufficient for interactive terminal-access to remote sites, was pushed

[10] also known as 'queuing system'

forward by quickly expanding capacities and bandwidths. At present it is capable to transfer huge amounts of data in reasonable time [27] and thus to crosslink all available resources, whose sizes may range from small few-CPU clusters to large arrays of thousands of individual nodes, such that computing power can be harnessed remotely. As a consequence only a few groups operate and fully exploit powerful computer systems on their own and instead share resources, because it is cheaper to access high computing power on demand without having to own the respective resources. The drawback of this sharing, however, is a mostly heterogeneous mixture of architectures, which have in common some Unix/Linux type operating system [28] but are completely different in terms of their operating and queuing systems. In contrast to the submission of tasks to single cluster sites, which are supported by a cluster management software, it is still an ongoing issue to provide suitable solutions to simplify the use of scientific applications across different domains and sites.

1.2.3 Origin and Evolution of the Grid

The more resources become part of a pool of computing facilities, the more complex and time-consuming becomes its administration for the user. Time, which a scientist is usually not willing to spend. To alleviate the use of distributed resources, it is highly desirable to design applications in such a way, that the user does not have to care where and how the calculations are performed. A concept pointing in this direction has already been anticipated in the seventies, at a time, when the very basic protocols for the Internet have been developed. Leonard Kleinrock [29] vaguely predicted in 1969 the evolution of a large infrastructure, which is capable of providing computing power in the same convenient way as the electric power-grid delivers electricity. The idea of a highly transparent and abstract network of computers is still a vision, yet the use of distributed resources has already evolved from a simple manual interaction by the researcher to a framework of protocols, automated mechanisms and aggregates of tools, which can be integrated by application developers to run tasks across large resource pools. This concept of harnessing and sharing distributed compute power was first termed ’meta-computing’ in 1992 by Catlett et. al. [30]. Similar to the electric power-grid, these new meta-computing infrastructures shall provide access to pervasive collections of compute-related resources and services [31], which finally coined the term ’Grid Computing’ in 1999 by Foster et. al. [32]. Today in general it may be defined as *“distributed computing performed transparently across multiple administrative domains, referring to any activity involved in the processing of digital information”* [33].

At the departmental level grids are built from workstations or commodity clusters which would otherwise be infrequently used but can be harnessed to obviate the need to purchase mid-range servers [18]. At the national level they may be built by collaborating computing centres or university research groups [34, 35, 36]. Activities at the international level involve government laboratories and large national centres [37, 38]. ’Grid Computing’ is understood as an advanced form of distributed computing and provides a large array of tools in order to use physically remote computing sites in the same way as local resources. In most approaches, the scientific part is separated from the computational one implying a certain layer in between the hardware and the application, which became known as the ’middleware’ [31, 39, 40, 41]. Unfortunately the different understandings of the purpose [42] of such a middleware, its capabilities and complexity lead to the development of numerous different products [43], which are briefly reviewed in section 1.2.4. A literature-based definition of ’Grid Computing’ is given in the appendix on page 162).

1.2.3.1 A scientists view on Distributed Computing

These many different and sometimes also contradictory definitions of a Grid are irrelevant to the scientist, since his perspective on computational Grids is usually quite simple [28] and driven by necessity. Hence, all usable[11] hardware shall be cross-linked with fairly simple tools , that can be applied by the scientist himself without special expertise on Grid-computing. Simplicity [44] and necessity [28] govern, that these tools do not have to provide all of the cited features [45], but only the most basic features to run applications on the many different computing resources, while most of the involved administrative tasks are done automatically in the background. One feature, which can mostly be omitted is the support for parallel calculations across many remote sites [46]. Given the definitions in literature, it is ’distributing computing’, what many applications really need. Therefore the capabilities provided by the projects cited in the following section would be of little relevance for most material science applications, since only a small fraction of their features can be used.

1.2.4 Major Grid-projects

Certain projects are spear heading the Grid-community by size and reputation. Their work has already had a significant impact on e-science [47], and therefore they are briefly introduced and described in the following sections.

1.2.4.1 GLOBUS

The GLOBUS project [48] is developing a basic software infrastructure for computations that integrate geographically distributed hardware and information resources. GLOBUS is a joint

[11]“Usable” does not include handheld devices, sensors or other electronic devices, whose main purpose is different to providing computing capacity. Hardware for data management exceeding the scope of a simple fi le-server is also not affected.

project of Argonne National Laboratory and the University of Southern California's Information Sciences Institute. The research project developed a software toolkit addressing key technical problems in the development of Grid infrastructures, services, and applications and provides technologies in a modular fashion being available under a liberal open source license. The

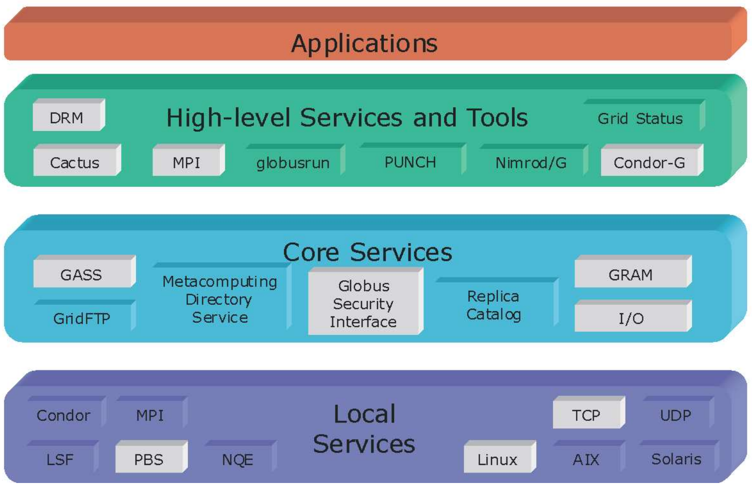


Figure 1.5: GLOBUS layers(Figure taken from [49])

layered architecture of GLOBUS is illustrated in Fig.1.5. Application developers are supposed to integrate high-level services into their software[12], which use the core-services to attach to a local architecture. One of its core components is the Grid Security Infrastructure [50] (GSI) for authentication and related security services. It provides public key based single-sign-on. GSI supports proxy credentials, interoperability with local security mechanisms, local control over access, and delegation. A wide range of GSI-based applications has been developed, ranging from SSH and FTP to MPI and other Grid-middleware [18]. The GSI is the most widespread component of the GLOBUS toolkit. Another essential component is the Grid Resource Allocation Management (GRAM), which allows programs to be started on remote resources. The Resource Specification Language (RSL) is used to describe the requirements of the application [51, 52]. The process of running applications in a GLOBUS environment and the interplay of its components and protocols is illustrated in Fig.1.6. Each component may be used independently or in connection with the other services. Applications need to include the corresponding component-libraries, which are provided among other languages in C, Fortran and Java. The installation however requires root-permissions. At present, the GLOBUS toolkit is available as version 4 [54], although the previous versions are still maintained and used by application developers [55].

[12]libraries and tools are provided as a Software Development Kit (SDK) in source code and binary form.

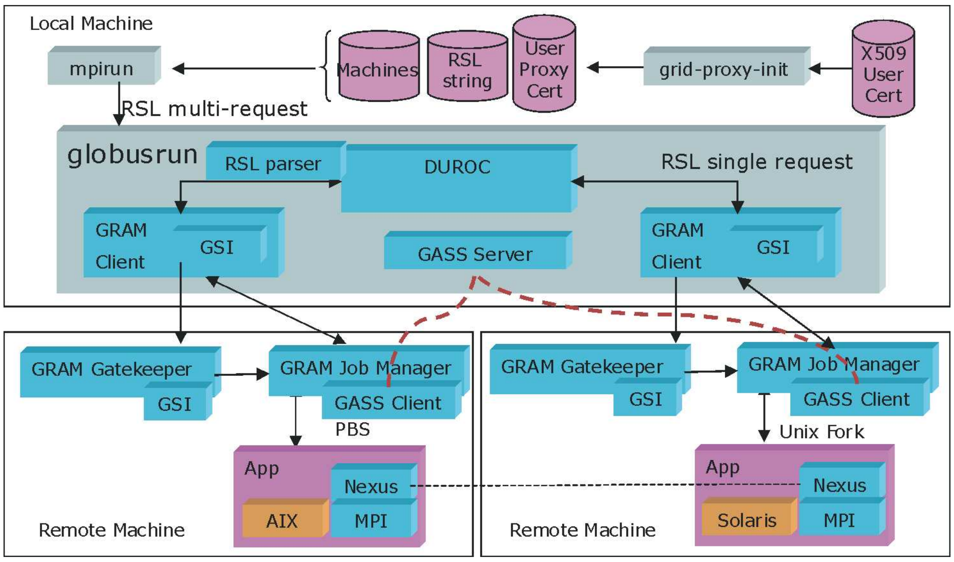


Figure 1.6: GLOBUS components (Figure taken from [53])

1.2.4.2 CONDOR

CONDOR is a specialised workload management system for compute-intensive jobs. The goal of the project is to develop, implement and deploy mechanisms that support high-throughput computing on large collections of computing resources with distributed ownership [18]. Like other full-featured batch systems, CONDOR provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to CONDOR, which places the jobs into a queue, chooses when and where to run them based upon a policy. It carefully monitors their progress and informs the user upon completion [56]. CONDOR can be used to manage a cluster of dedicated compute nodes (such as a "Beowulf"cluster [57]). In addition, unique mechanisms enable CONDOR to effectively harness wasted CPU power from otherwise idle desktop workstations. For instance, it can be configured to only use desktop machines where the keyboard and mouse are idle. Should CONDOR detect that a machine is no longer available (such as a key press detected), in many circumstances it is able to produce a checkpoint and migrate the job to a different machine. CONDOR does not require a shared filesystem across machines, it can transfer the job's data files on behalf of the user, or it may be able to transparently redirect all the I/O requests of a job back to the submit-machine. As a result, CONDOR can be used to seamlessly combine all of an organisation's computational power into one resource [56] (CONDOR pool). Several run-time mechanisms are provided in the CONDOR model to facilitate different load sharing strategies. The purpose is to harness CPU cycles from idle workstations [58, 59]. However, several essential components require root-permissions.

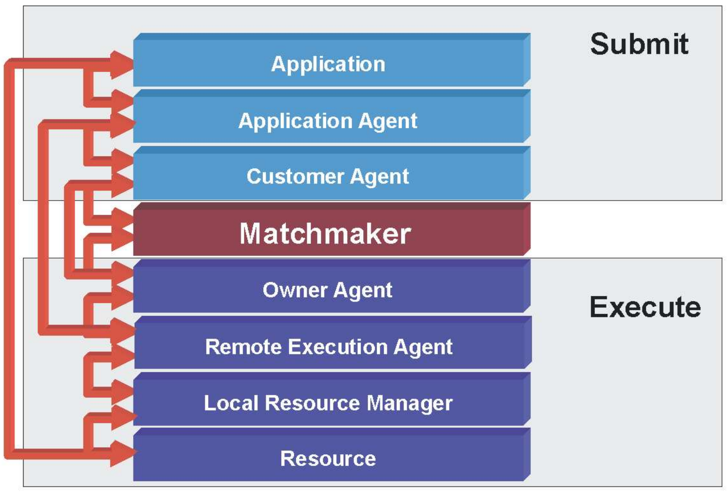


Figure 1.7: CONDOR Layers (Figure taken from [56])

1.2.4.3 UNICORE

The name stems from Uniform access to Computing Resources and is funded by the German ministry for science and education, starting in August 1997, as a prototype for sharing access to facilities at German supercomputing centres intended as a "ready to use"alternative for the GLOBUS toolkit [18]. The aim is to provide seamless, secure and intuitive batch access for diverse computing resources [60, 61]. The architecture consists of three layers, namely user,

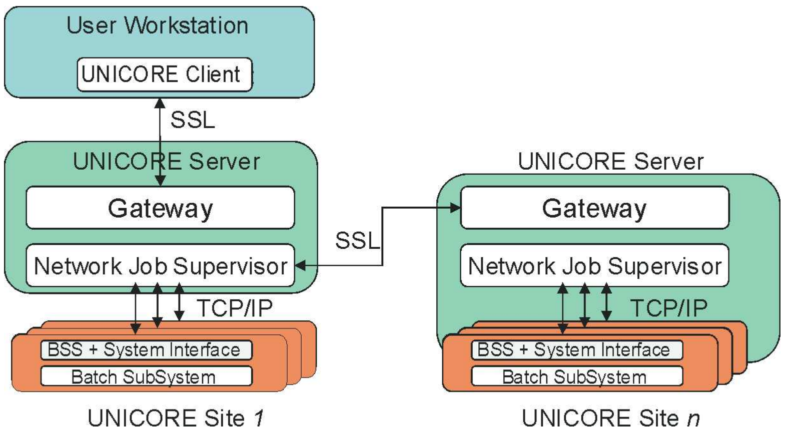


Figure 1.8: UNICORE layers (Figure taken from [62])

server, and target system. The user is represented by the UNICORE Client, a Graphical User Interface (GUI) that exploits all services offered by the underlying server layer. Abstract Job Objects (AJO), the implementation of UNICORE's job model concept, are used to communicate with the server layer. An AJO contains platform and site independent descriptions of computational and data related tasks, resource information, and workflow specifications. The

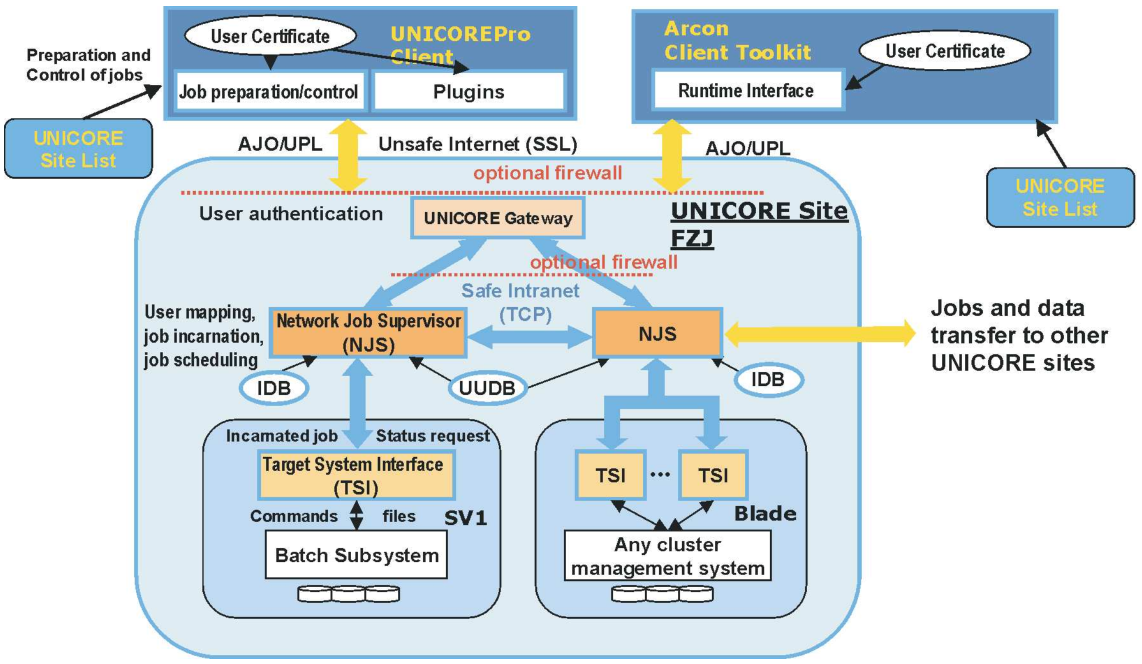


Figure 1.9: UNICORE architecture (Figure taken from [62])

sending and receiving of AJOs and attached files within UNICORE is managed by the UNICORE Protocol Layer (UPL) that is placed on top of the Secure Socket Layer (SSL) protocol. The user of an UNICORE Grid does not need to know how these protocols are implemented, as the UNICORE Client assists the user in creating complex, interdependent jobs. For more experienced users a Command Line Interface (CLI) is also available. Both, the UNICORE Client and CLI, provide the functionalities to create and monitor jobs that can be executed on any UNICORE site (Usite) without requiring any modifications, including data management functions like import, export, or transfer of files from one target system to another. In addition, a plugin technology allows the creation of application-specific interfaces inside the UNICORE Client [63]. The middleware includes a web-based Java GUI for batch submission, which facilitates the distribution of tasks to the most suitable platform and site. Information about resources is provided. Use is made of existing technology with access to distributed data. The three-layer approach comprises a browser running on the user's workstation that communicates with a UNICORE gateway running at any of the collaborating sites. This contains an authentication procedure (using X.509 certificates) and site-specific authentication and login authorisation. Finally a resource management layer will submit the job to the local system or initiate further authentication for submission to a remote site. Early users included Debis and INPRO who carry out modelling work for the German automobile industry. Certain components must be installed as root, but for the majority of components user-permissions are sufficient. Many of its features are built on GLOBUS components [64].

1.2.4.4 NETSOLVE/GRIDSOLVE

It is an RPC based client/agent/server system that allows one to remotely access both hardware and software components to harness loosely coupled systems on a network [18]. The purpose of GRIDSOLVE is to create the middleware necessary to provide a seamless bridge between the simple, standard programming interfaces and desktop Scientific Computing Environments (SCEs) that dominate the work of computational scientists [65] and the rich supply of services supported by the emerging Grid architecture, so that the users of the former can easily access and harness the benefits (shared processing, storage, software, data resources, etc.) of using the latter [66]. NETSOLVE is intended to provide transparent access to a whole variety of software libraries, highly tuned for the target architecture. This improves maintainability of software and avoids the end user having to download and compile it. NETSOLVE is implemented as a three-tiered system [18]:

1. The **client** may be a C or Fortran program linked to the NETSOLVE library, Mathematica sessions, or Java applets calling the NETSOLVE library. If the client needs resources to solve a computationally demanding problem, this request is sent to the agent, which performs a match-making and selects a proper resource from the pool of servers.

2. **Agents** are C programs running as daemons and act as resource brokers.

3. **Servers** are registered with agents and can perform certain services (e.g. have particular applications installed). They are supposed to provide an optimal computation environment for their particular architectures. A server processes the task and returns the results to the client.

If the client needs resources to solve a computationally demanding problem, this request is sent to the agent, which performs a match-making and selects a proper resource from the pool of servers. The server finally processes the task and returns the results to the client.
At the API level NETSOLVE looks like a high-level library with a single function call (**'netsolve'**). Character strings are introduced to specify the required action. A non-blocking version is also available, but the user has to take care of resource usage and determinism [67, 68]. NETSOLVE searches for computational resources on a network, chooses the best one available, solves a problem (e.g. matrix-multiplication), and returns the results to the user. A load-balancing policy is used by the NETSOLVE system to ensure good performance by enabling the system to use the computational resources available as efficiently as possible. The framework is based on the premise that distributed computations involve resources, processes, data, and users, and that secure yet flexible mechanisms for cooperation and communication between these entities is the key to meta-computing infrastructures. Interfaces in Fortran, C,

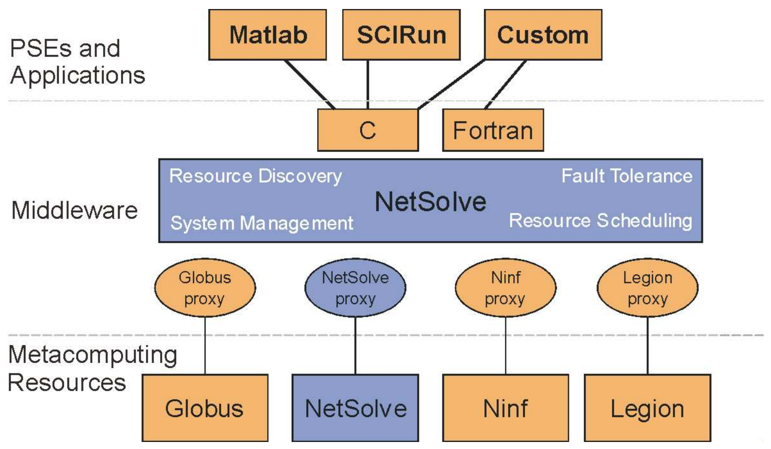


Figure 1.10: NETSOLVE architecture (Data for the fi gure taken from [66])

`Matlab`, **`Mathematica`**, and **`Octave`** have been designed and implemented which enable users to access and use NETSOLVE more easily, it currently has an interface to ScaLAPACK and related components [66]. NETSOLVE uses CONDOR for its distributed computing management.

1.3 Diffi culties with existing middleware

The majority of the existing toolkits and middleware projects have certain peculiar handicaps, which limit their applicability for scientific programs. Especially small workgroups and single researchers can only poorly benefit from the offered solutions due to an excessive overhead for the implementation and the installation on the computing sites. Such and other problems have been identified by numerous authors [28, 45, 69] and shall be summarised below.

- The existing toolkits have an excessively heavy set of software and administrative requirements, even for relatively simple demands from applications.
- Currently propagated 'general' solutions require that someone dedicates a lot of time to the implementation.
- They are mostly painful and difficult to install and maintain, due to excessive reliance on custom-patched libraries, poor package management, and severe lack of documentation for end users.
- Middleware developers only poorly cooperate with application developers and therefore run a substantial risk of producing and implementing Grid architectures which are irrelevant to the requirements of application scientists.

- The development of protocols and standards are driven by a few large scaled communities [38], which will only poorly accommodate the present hardware situation of small research groups.
- Frequent changes of paradigms, protocols and interfaces as well as very short compatibility periods make it difficult for application designers to implement their software for use with a certain toolkit, since this is a major effort and a strategic decision for many years. As a consequence, there is a significant lack of real Grid-applications [70], because scientists rather stick with the technology they know and from which they can expect to have an adequate life cycle.
- The offered solutions do not consider the typical hardware situation of scientific workgroups, which is a mixture of accounts at Computing centres, some own clusters, several Desktops and numerous 'borrowed' resources [28]. Very different among all the individual items is the security policy, the policy of administration and the level of permission.

The personal resource pool of a researcher spans several sites with different tools and maybe certain Grid-infrastructures provided, but it is highly improbable that all different administrative domains offer the same - if any - middleware. Unfortunately the powerful, yet heavy-weight common Grid-middleware requires permissions a user mostly is not granted, especially on the most powerful clusters. It is therefore impossible, that he may install any of the Grid infrastructures listed in section 1.2.4 by his own means. As a consequence he either has to abandon several resources and limit himself to the few matching architectures, or employ a different middleware, which has fewer requirements.

1.3.1 Implementation paradigms

Heavy-weight infrastructures do not only limited the use of computing sites. Their disadvantage also becomes evident in the case of application design, since there are two different methods to make an application run on distributed resources.

- The necessary functionality to access the Grid-resources are **'embedded'** into the program by libraries (C, Java, Fortran, etc.). This invasive approach requires mostly vast changes to already existing applications. Porting the application is difficult, due to the numerous libraries, which are required in advance. Hence the result is prone to compatibility problems. The undeniable advantage, however, is an optimal performance.
- The non-invasive approach only wraps an application (mostly) by the aid of scripts in any high-level language and **'attaches'** Grid-capabilities. The scripts bridge the gap between the application and the Grid and require a proper set of interfaces (files, commands) for

steering. The application will remain almost unchanged, but it cannot beat the invasive method in terms of performance. The implementation is usually accompanied with less effort and offers a better portability.

Numerous scientific applications are already in use since many years and have originally been designed to run locally, maybe in parallel but not in a distributed environment. An invasive approach will usually require to redesign the applications, which is an effort only few developers are willing to take, especially if the application performs quite well on local resources, therefore the most favourable approach is the non-invasive one, since it will leave the original code mostly untouched.

1.3.2 'Lightweight middleware' as a solution

The numerous problems arising from the use of such powerful toolkits inspired a plea [45] for lightweight middleware [28, 71, 72]. The common objective is to abandon several features, which may only be interesting for certain **global-scale** [38][13] but not for **small-scale** projects, in favour of a simplified installation and an easy implementation. It is desirable for lightweight solutions to maintain a reasonable relationship between the effort of implementation and the yielded benefits, hence there are a few important requirements:

- The middleware should be easy to install without root-permission to machines.
- It must be substantially more portable, lightweight and modular in design as well as being extensible with little effort.
- It should provide sufficient safety to convince system administrators to install and use the middleware on their site.
- The installation of additional software/libraries/modules should be avoided. The infrastructure shall make use of tools, which can be expected to be available on every common Unix/Linux based system.
- The existing scientific application should not be changed in any way, but instead only wrapped by proper scripts in a non-invasive way.

The concept of lightweight approaches found approval by many scientists and developers and is already applied to several products [69, 73, 74, 75, 76, 77, 78].

[13]like the 'LHC computing grid', which runs a complex and time-consuming task in parallel on many thousands of different nodes at the same time and accesses Terabytes of data from many physically remote databases [69].

1.4 W2GRID

The development of **W2GRID** should based on the demands for a lightweight middleware and hence the installation of all parts - server-side and client-side - must be possible exclusively with user-permissions and should be manageable with little effort by the user without special expertise. Additionally the whole middleware must be a standalone product, that is independent of any third-party software (libraries and applications) and only relies on the tools and programming languages, which can be expected on any common Unix/Linux platform ("standalone principle"). It should be allowed, that each user may create his "personal Grid" (Fig.1.11) by linking all the various computing resources he has access to, while at the same time this Grid may overlap with the "personal Grid" of other users, who have a different resource pool. Therefore it should also be possible, that individual user-installations of **W2GRID** run independently on the same host. Implementing applications with **W2GRID** must be possi-

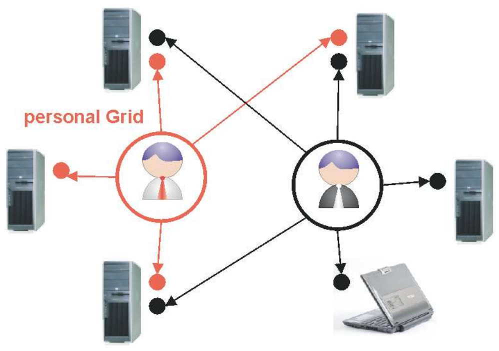


Figure 1.11: A user's personal Grid as provided by **W2GRID**

ble in a non-invasive way with the aid of scripts, which just wrap the application thus avoiding to change the source code. These scripts must be portable, such that both, the user and the developer do not need to hard code any platform or job-submission specific commands into the scripts. In order to simplify the interaction of the application with the infrastructure, it should be possible to completely separate the job-submission from the application, such that it is sufficient to specify once (during installation) how the job-submission works in general on a given host. Then all application specific scripts can simply use these definitions, without having to contain any hard coded host-specific commands. None of the other lightweight projects (see section 1.4.3), could fulfil all the demanded properties, though some approaches and solutions come close but lack the desired independence from any kind of third party software, which is demanded for **WIEN2k** and material science applications as a whole.

1.4.1 Design philosophy

`W2GRID` should be implemented as a client/server infrastructure and should make use of remote-procedure-calls (RPC) [79, 80], which have been found to be powerful abstractions for distributed computing [81, 82, 83]. This approach is particularly useful since it provides an intuitive programming interface, allowing users to easily make applications Grid-enabled [84] by the aid of 'commands', which may be used in a similar fashion as local ones. For portability the code of **`W2GRID`** has to be separated into two parts:

- A fully portable **core**, which does neither contain hard coded routines for individual platforms, methods of job-submission, file- and data transfer nor application-specific commands.
- An array of **plugins**, which contain the specific capabilites for the listed purposes. The plugins can be included into the code on demand and are independent from each other. All capabilities provided by these special kind of libraries are used (in an abstract way) by other plugins and the elements of the core.

The application-specific scripts must come as plugins too (referred to as "application plugins"). The advantage of this modular approach is, that all kinds of plugins, once they are developed, can be shared and used without further adaptations. As a result of this concept **`W2GRID`** does not need to integrate any third-party Grid-components such as the GSI [50] of the GLOBUS toolkit (section 1.2.4.1), but it may interoperate with other middleware and their tools by means of a proper plugin.

1.4.2 Desired benefits for WIEN2k

Licenses for **`WIEN2k`** have already been sold over a thousand times worldwide to workgroups and companies, which use their specialised hardware configuration and maybe their very own Grid-middleware. The **`W2GRID`**-implementation must remain an optional feature, which can be attached to the existing application, but **`WIEN2k`** must remain operable without **`W2GRID`**. The workflows coded into the application plugin must help the user to handle the complex filetransfer of the numerous input- and output files involved in a calculation. Additionally it should support the non-trivial creation and control of coarse grained parallel processes, which require a monitoring and automated steering for performance optimisation. As a whole, the plugin must help to improve the usability of **`WIEN2k`**.

1.4.3 Related work

Some of the related projects have been developed at the same time as **W2GRID**, and are described because of the similar approach in terms of simplicity. What all the projects have in common is that they can be installed with user-permissions and provide the applications and additional functions by means of web-services.

1.4.3.1 Web-Service based Environment for Distributed Computing (WEDS)

WEDS is a hosting environment for distributed simulations within a single administrative domain. It is designed to let scientists remotely deploy single or multiple instances of a pre-

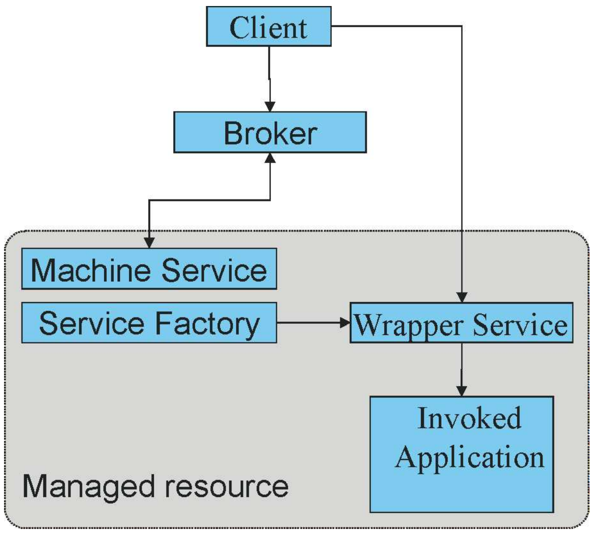


Figure 1.12: WEDS architecture (Figure taken from [74])

existing code across multiple resources and give steering, visualisation and workflow functionality with only simple modifications to the program code. WEDS [85] is built on a lightweight Perl-based WSRF compliant web services container (WSRF::Lite), developed at the University of Manchester. WEDS is written in Perl and is easy to install by ordinary users, requiring little effort on behalf of the system administrator. Only standard libraries (Perl, SOAP etc) are required for a successful implementation and can be installed in a user specified directory. WEDS requires at least two ports. It was developed and produced by scientists, and intended to be used for simulation launching, interaction and visualisation. WEDS is suitable for low-complexity Grids, for which a single broker machine can manage all registration and job-submission tasks [74].

1.4.3.2 Vienna Grid Environment (VGE)

VGE is a prototype framework for Grid-enabling HPC applications, which can be exposed as generic application services and securely accessed by multiple remote clients over the Internet within a service-oriented environment. VGE has been realised based on state-of-the-art Grid and Web Services technologies, Java and XML [86]. As a key feature, VGE supports a flexible QoS[14] negotiation model, which guarantees on execution time and price with potential service providers. The VGE service provision framework is currently utilised in the context of the EU Project GEMSS, which focuses on the provision of advanced medical simulation services by means of a Grid infrastructure [78]. The VGE is composed of a Grid Service Environment (VGSE) and a Grid Client Environment (VGCE) (see Fig.1.13).

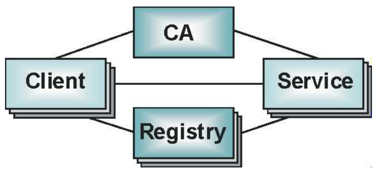


Figure 1.13: VGE architecture (Figure taken from [86])

- VGSE is an application framework intended to be used by service providers in order to ease the process of transforming native applications into Grid Services. It is available for Solaris 9, Windows 2000,XP,NT and Linux.
- VGCE is a client-side framework and API for the development of user interfaces for Grid-enabled applications. VGCE supports the selection of and interaction with Grid services from a client-side application component or a user interface.

1.4.3.3 Application Hosting Environment (AHE)

The AHE [77] is designed to provide the simplest possible service interface to a client for submitting jobs to highly complex grids. The AHE, similarly to WEDS, is a lightweight web service hosting environment but is able to operate over multiple administrative domains. The AHE stores all necessary informations about how an application should be run on the various computational resources of a Grid and provides uniform interface to the client for running that application across those resources. The AHE can interact with the GLOBUS toolkit and CONDOR. Its design assumes that the client is behind a firewall allowing only outgoing connections. All the AHE requires from the client is to support HTTP HTTPS and SOAP. The lightweight client could be accessed by the user via a PC, a PDA or even by mobile phone. It provides

[14]Quality-of-Service

a higher level abstraction of a Grid than is offered by existing grid middleware schemes such as the GLOBUS toolkit. As a result the computational scientist does not need to know the details of any particular underlying Grid middleware and is isolated from any changes to it on the distributed resources. The functionality provided by the AHE is ‘application-centric’: applications are exposed as web services with a well-defined standards-compliant interface. This allows the computational scientist to start and manage application instances on a Grid in a transparent manner, thus greatly simplifying the user experience [74].

1.4.3.4 Styx Grid Services (SGS)

SGS is based on web-services and focuses on commandline programs, which are wrapped and may be used exactly as if they were installed locally. Workflows may be created on the basis of shell-scripts by simply invoking remote services from a general-purpose commandline. Each service will establish a persistent connection until the task is completed. The software is lightweight and quick to install[15]. There are only few demands on firewalls, only one incoming port to the server is required and no incoming ports to the client machines. The key idea of SGS is, that all resources are represented as files, hence it is a kind of file sharing protocol and therefore every service can be represented by its URL. Programs are wrapped by an XML description, specifying the commandline parameters and input files as well as the output files. Remote programs are started from a general-purpose commandline client, which fetches the XML description from the server, parses the data and the arguments to the commandline interface, uploads the input and starts the remote program. The SGS-protocol does not require any particular security mechanism [76].

1.4.3.5 Other Concepts

Myers et.al. developed a client/server model, which employs simple worker-clients on remote resources and a more complex server at the local site, according to the idea, that at the local site root permissions will be granted, hence the processes running there can be allowed to be more complex, whereas the single clients are sufficiently served with any Unix/Linux architecture, Perl and some standard tools [28]. GRIDBENCH [87] uses a plugin-concept for interoperability with Grid middleware in order to perform benchmarks. Application plugins can also be found in NETSOLVE [66].

[15]less than 5 MB

Chapter 2

Implementation and Concepts of W2GRID

2.1 Programming languages

Every programming language can be characterised by its specific features, advantages and disadvantages. In general there exists a simple relationship: The more powerful, the more complex is the development of code. Hence this may lead to an unwanted overhead if the resulting program shall only serve a simple purpose (i.e. an installation routine). A software like **`W2GRID`**, which consists of many independent components with different purposes and demands should therefore not be written by the use of only a single programming language. In contrast it will profit from combining the best facets of a few different ones, to satisfy the three predominant purposes, which are listed below.

- **Data manipulation:** time-critical read/write operations from and to files. The respective programs should have minimal memory consumption and provide an optimal performance.

- **Network components (i.e. daemons):** as can be seen in Fig.2.1, **`W2GRID`** will need some highly flexible network components which should not require an excessive effort for development and debugging. These daemons in general perform complex tasks and will consist of dynamic code, which may change very often. The performance - although important - is less critical than the stability (robustness of the infrastructure). Regular expression handling will be needed to extract data from the output of numerous tools.

- **Installation:** will consist of standard Unix/Linux commands embedded into simple syntactical elements such as conditions and loops. The implementation should be possible with the least effort. Issues like performance or memory consumption can be neglected.

For each of the given demands, an individual programming language has been selected, which is supposed to serve the respective purpose best. Those, which are finally used for the implementation of **W2GRID** are chosen from among several equally or even better qualified ones, due to an important constraint: It is demanded that the compiler/interpreter must be available and already pre-installed on most platforms. Therefore popular ones like Java are omitted in favour of a more widespread one. Finally it is taken into account, that the resulting code needs to be maintained by the members of the workgroup, hence the preferred language is also the most frequently used one in the group (e.g. csh vs. bash; see below). The provenience, advantages and disadvantages of the three chosen ones are listed as follows.

2.1.1 C

C is a general-purpose, procedural, imperative computer programming language developed in the early 1970s by Dennis Ritchie for use on the Unix operating system. Since then it has has spread to many other operating systems, being now one of the most widely used programming languages. C is commonly used in computer science education, in part because the language is so pervasive[1]. It is applied for complex and time-critical file I/O operations of **W2GRID**, namely for the wiensql-database (see section 2.5), which is needed by most other components. It has to provide optimal performance and avoid bottlenecks

- **Advantages:** Yields very fast and powerful applications with minimal memory consumption. This is especially important for the wiensql-database, because there will be many simultaneous instances.
- **Disadvantages:** Difficult to debug, since the data-types and also the memory allocation must be handled explicitly by the developer. Code, which is subject to frequent changes but less critical in terms of memory consumption and performance is mostly better served with a more flexible yet equally powerful language (e.g. Perl or python).
- **Alternate choices:** Perl is significantly more demanding with respect to its memory consumption and is also inferior in terms of the performance of file-operations. It can therefore not be used for the wiensql-database. C++ is more advanced, hence more powerful and almost equally widespread, it is however not used for **W2GRID**, since most C++ compilers do not completely support ANSI C++ standards. Furthermore, the capabilities of C are sufficient.

[1] http://en.wikipedia.org/wiki/C_programming_language

2.1.2 Perl

The language has originally been created by Larry Wall in 1987 and is now a de-facto standard, which can be expected to exist on any Linux/Unix based distribution[2]. **W2GRID** requires at least version 5, since it makes use of references and modules, which do not exist in earlier releases. Perl offers the needed capabilities to develop powerful network components but is much easier to implement than the corresponding C-code. It is used for the two daemons, which are referred to as 'the backbone' of the **W2GRID** infrastructure (see Fig.2.1). Perl offers the desired flexibility for the frequent changes, updates and adaptations.

- **Advantages:** The language handles the data-types and memory allocation internally, hence the developer does no have to care about these issues explicitly. Thus the sources are interpreted and don't need to be compiled, which simplifies debugging. Only little effort is needed for creating daemon processes, yet it is highly portable and very powerful, because it borrows features from C, Bourne-Shell (sh), ***awk***, ***sed***, Lisp, and, to a lesser extent, many other programming languages. Finally it comes with the advantage, that additional code can be included at runtime without previous compilation or linking (see the concept of core and plugins in section 2.9). Regular expression handling is a built-in feature, which makes string-manipulations one of its core competences.

- **Disadvantages:** Creates large processes in memory and requires, that the application is thoroughly checked for memory leaks.

- **Alternate choices:** Java and Python are not available by default on all different platforms yet, especially not on older distributions. Other scripting languages are still more exotic and less widespread. Most of them would need to be installed by the user in advance to the **W2GRID**-installation, which would corrupt the standalone concept (see 1.4).

2.1.3 C-Shell (csh)

Being a very convenient and straightforward scripting language, which is most common on almost every platform, C-Shell is chosen for the implementation of the **W2GRID** installation routines. It was developed by Bill Joy for the BSD Unix system, but originally derived from the 6th Edition Unix ***/bin/sh*** (which was the Thompson shell), the predecessor of the Bourne shell. Its syntax is modelled after the C programming language. Nowadays the C-Shell[3] exists on almost all Unix/Linux operating system distributions. If used as login-shell, it is often replaced by the Tenex C-Shell (tcsh), which offers more capabilities but basically has the same syntax.

[2]http://en.wikipedia.org/wiki/Perl
[3]http://en.wikipedia.org/wiki/C_Shell

- **Advantages:** Provides a simple implementation of an installation workflow (e.g. creation of directories, expanding archives, copying files). Error-handling and debugging at the console-level is easy. It is highly portable, if only standard Unix tools and their standard flags and options are used (e.g. ***ps***, ***grep***, ***sed***, ***awk***, ***top***, ***tar***, ***gzip***, ...).

- **Disadvantages:** Different to other shell-scripting languages, csh uses an inconvenient way to implement subroutines, which employ the 'alias' command. Especially large programs (i.e. several hundred lines and more) may therefore be difficult to read. The pattern matching with ***grep***, ***egrep*** and ***sed*** is cumbersome and can better be accomplished by Perl.

- **Alternate choices:** The Bourne-Again Shell (bash) is equally suited for the same purpose and even provides functions, however its syntax is less C like and it is not as frequently used by the workgroup members, so that it was omitted in favour of the csh. Other shell-scripting languages like the Korn-shell (ksh) are not as widespread, Perl on the other hand is too complex for most of the simple tasks, which need to be performed during the installation. Additionally it lacks a convenient and portable[4] method for erasing the content of the terminal window (e.g. ***clear*** of csh).

2.2 Client/Server architecture

`W2GRID` is intended to turn a number of individual hosts into a transparent environment. According to the definitions of Orfalie et.al [89], the server 'exports' capabilities, which are requested by the client. This architecture-style is widely applied for distributed computing [90]. **`W2GRID`** will make use of Remote-procedure-calls (RPC) for this purpose (see 2.6.2.1) and accordingly employ two Perl-daemons: The **GridServer** provides access to the computing resources and has to be installed on each host. The **GridClient** on the other hand will manage the communication between the user and his resources and needs to be installed only once on a single site (i.e. the local desktop). The left-hand side of Fig.2.1 - the **User-side** - is in most cases the local desktop, whereas the right-hand (or **Server-**) **side** represents each individual resource[5]. The illustrated items of Fig. 2.1 are briefly discussed in the following list.

[4]Some modules (e.g. Term::InKey) are available on CPAN [88]. Because this is third party code, none of them is used.

[5]For clarity, the third daemon (wiensql-database) is omitted in this fi gure. It is the central tool for data management of all other components of **`W2GRID`**, the GridServer, the GridClient and several Perl-tools and will be explained in section 2.5.

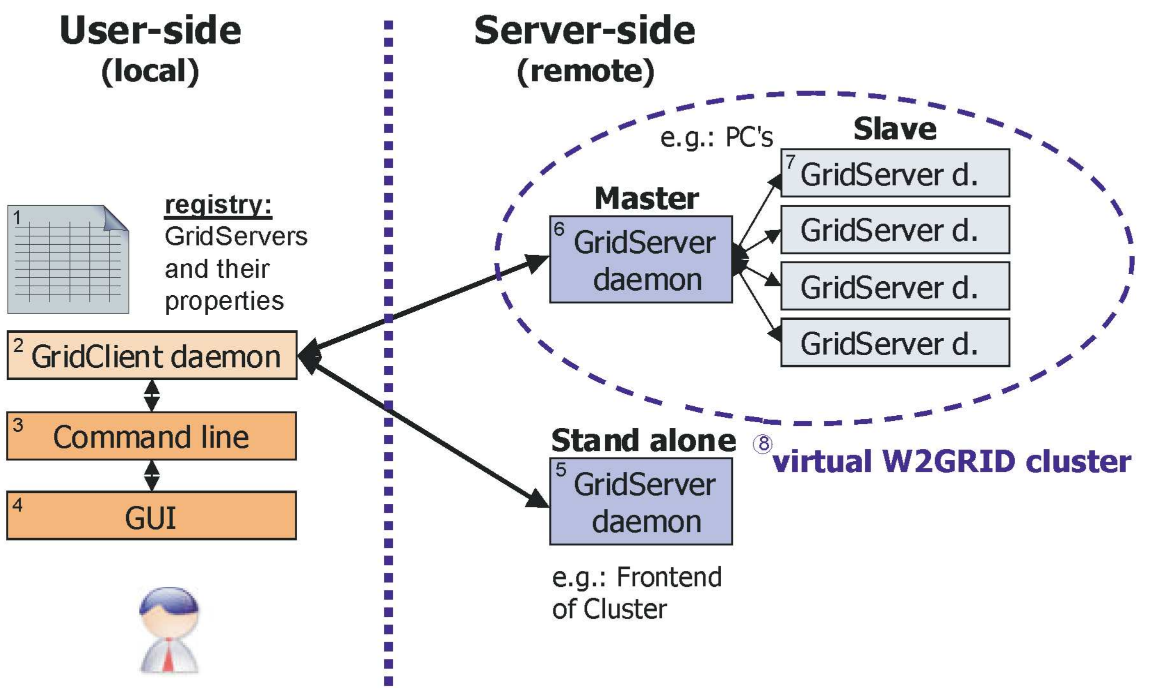


Figure 2.1: **W2GRID** architecture

1 The registry is a database-table provided by the wiensql-daemon and contains a list of GridServers and the associated informations of how to connect to them. It will be explained in section 2.6.6.

2 The GridClient (section 2.2.2) represents an interface between the user and his GridServers.

3 **W2GRID** provides a simple commandline interface for the user in order to interact with the GridClient daemon. It offers a similar usability like a common terminal client and may also be used in batch mode (see 3.2.3)

4 By embedding such batch mode calls to the commandline interface into dynamic HTML-code[6] one can obtain a web interface [91], but **W2GRID** does not provide such a GUI [7] yet.

5 A GridServer (see section 2.2.1) is mandatory on each resource. When installed on a managed cluster, served by some queuing system, it only needs to be present on a single node (usually the frontend). If used for desktops, which are not clustered by cycle-

[6]PHP, Perl, etc.
[7]Graphical-User-Interface

harvesting software such as CONDOR [58], the GridServer has to be installed on each single desktop

6 Each of these desktop GridServers can be used as a one-node resource, but a high number of individual hosts may be difficult to manage. Therefore **`W2GRID`** offers to emulate the functionalities of a queuing system by the use of an additional setting, which is applied during the installation. The Master will serve as the frontend, whereas the Slave 7 represents an individual node of the **`W2GRID`**-'cluster' 8, which is only 'virtual', because the middleware lacks the proper permissions to manage the resources in the same way, a queuing system may do this. Thus it lacks the ability to guarantee any quality of service.

The presented infrastructure supports only one-way connections from the client to the server [92]. A 'call-back' in the other direction is not possible, hence the GridServer cannot broadcast any event (e.g. the completion of a calculation) to the GridClient. Instead the latter needs to connect in regular intervals and request the corresponding informations (e.g. the status of a calculation).

2.2.1 Purpose of the GridServer

As it was already said above, this Perl-daemon is installed on each remote computing resource and serves as a 'frontend' for job-submission and system-status retrieval (e.g. the current load, the free memory, list of running processes). Especially the job-submission is a critical and demanding task in distributed computing, since this is very individual among the different hosts and requires the exact knowledge of the local features (e.g. the details for writing a submission-script) and the details of the given installation (e.g. the way how additional tools like MPI can be applied). A sample code for such a submission script on a LoadLeveller-managed cluster is given in Fig.A.3 in the appendix. All processes invoked by the GridServer will run in behalf of the user, who owns the daemon. As a consequence, they will have only the same (user-)permissions granted.
The daemon can be set up in three different modes: The **standalone** mode (default) is used on the frontend of clusters or on individual desktops, whereas the **master-** and **slave-** modes allow to generate a virtual **`W2GRID`** cluster 8 $_{2.1}$. The GridServer will accept the input (and also return the output) in a machine-readable and platform-independent XML-format (see for example the output of Fig.2.2). Status data like the 'load' are returned in a transparent way, independent of any specific formatting due to the operating system or the job-submission scheme. Both daemons (the GridServer and the GridClient) are controlled by the aid of RPC[8]

[8]Remote-Procedure-Calls will be explained in section 2.6.2

```
<HOSTS>
   <NODE>
      <NAME>athena</NAME>
      <LOAD>0.25</LOAD>
      <WEIGHT unit=percent, type=average>25</WEIGHT>
      <QUEUED>0</QUEUED>
      <FREENODES type=total>1</FREENODES>
      <FREESLOTS>6</FREESLOTS>
      <FORECAST unit=percent, type=best>80</FORECAST>
      <STARTTIME unit=seconds, type=best>0</STARTTIME>
   </NODE>
</HOSTS>
```

Figure 2.2: Sample output of a 'load' request as returned from the GridServer

commands, which represent different workflows. To extend the capabilities, new workflows (i.e. new RPC-commands) must be developed (see section 3.3.1).
Direct interactive access to the GridServer-daemon is possible for debugging purposes by using the provided commandline interface '***gridsrv_console.pl***', yet this is not very appreciable due to the XML-format of input and output. The user, however, will not have to interact in person with each GridServer daemon, since this is done by the GridClient on his behalf (see below).
W2GRID allows that either one daemon can be shared among several users or that each user can run his private one. The individual daemons on the same machine will be independent from each other. However on a managed cluster 5$_{2.1}$ it is recommended to use individual daemons, whereas on a 'virtual cluster 8$_{2.1}$', several users should share a single daemon, as this simplifies the administration.

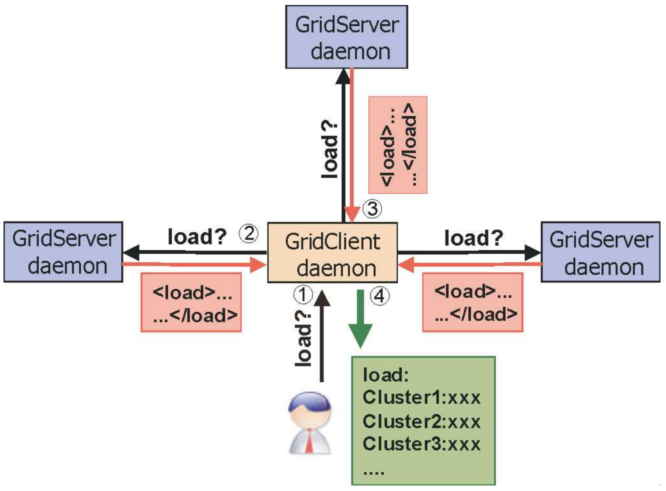
2.2.2 Purpose of the GridClient

Because the registered resources may be quite numerous[9], it would quickly become an annoying effort to contact each single one directly by the use of the commandline interface and issue the mostly complex commands. The GridClient 2$_{2.1}$ can do this on the user's behalf with much more efficiency. Additionally it serves as resource broker. Depending on the implemented workflow, the GridClient can search for the most suitable resource, transfer the input files and use the provided RPC-commands of the GridServer to submit tasks to the queuing system. Out of all components of the **W2GRID** infrastructure, the GridClient is the only daemon, which is contacted interactively by the user. Different to the GridServer interface, the input and the output (as sent and received by by the commandline interface ***gridclient.pl***) is therefore sent as plain text, as it is supposed to be read and written by the user. Every user needs only a single GridClient daemon.

[9]One GridServer for each resource

2.2.3 Interaction between the daemons and the user

Image 2.3 shows the communication in between the three main-entities in the **W2GRID** environment. A user submits a request (e.g. to check the 'load'-situation of his resources) 1 to the GridClient. Depending on the triggered workflow, this request (see section 2.6.2) will require, that several (or all) registered GridServers are contacted 2 (in parallel). Finally all the incoming results 3, which are provided in XML format for improved machine-readability are collected and merged into the final report. The output returned to the user 4 contains a summary of all retrieved data.



- ***black arrow* 1 *:*** *The 'load' situation is a dynamic information and must be retrieved on-demand, hence the request is 'forwarded' 2 to the GridServers.*
- ***red arrow/box* 3 *:*** *The intermediate results must already be provided in an architecture independent format. The 'load' for example is returned as a fraction of busy nodes of the total number of available ones.*
- ***green arrow/box* 4 *:*** *Since the intermediate results are XML-formatted, the GridClient must render the data into well-formatted plain text.*

Figure 2.3: The interaction between the user, the GridClient and the GridServer

2.3 Security

The **W2GRID** middleware is supposed to be installed on foreign resources, therefore it must provide a sufficient degree of safety, since system- administrators do not like having their firewalls or other security measures punctured from the inside. For this reason it is desired to offer a secure product, which can withstand the most common and threatening attacks [93].
The three daemons (GridServer, GridClient and wiensql), which are explained in section 2.2 employ similar security mechanisms. Hence they will be described here in general. The purpose of the security measures is to avoid that any component of the infrastructure is abused by an attacker to crash the host, block authorised users from accessing **W2GRID** or to hack the user-account and corrupt his data.

2.3.1 Policy

Much safety can already be provided by avoiding common risks. **W2GRID** is supposed to be installed and run by users with their respective user-permissions only, therefore none of its processes will have root-permissions granted and thus cannot endanger essential components of the architecture. Additionally the default RPC-commands (see section 2.6.2) will not offer any interactive terminal-access to the host, so even in the very unlikely case of an intrusion, these commands cannot be abused to perform illegal operations.

2.3.2 Encryption

Avoiding plain text for the communication between the daemons contributes to the safety, too. Even the intranet-connections should be secured by encryption, because all traffic can easily be ’sniffed’ by any host of the local network.

2.3.2.1 AES

The encryption is symmetric, in contrast to the very popular asymmetric “public-key cryptography”, which is applied by SSH for example. The same key, which is used for encryption also serves for the decryption. It is evident, that this key must be kept secret, hence the method is also known as “secret-key cryptography”. The algorithm yields an excellent performance, even for long texts. For details about the method it shall be referred to the literature [94, 95] and the appendix (160). The algorithm is not protected [96] as intellectual property and can be downloaded and used freely. It is amended to the **W2GRID** sources and used as a C-library (rijndael.o†). The encryption causes some delays to the communication, therefore it is only applied for the critical logon-phase (see section 2.3.4), whereas all traffic following the successful authentication of the client will be carried out in plain text[10]. For the sake of performance, the encryption can be turned off completely, which is not recommended.

2.3.2.2 SSH-tunneling

A remote port is accessed through a secure connection[11].This method has two advantages: On the one hand the entire traffic (not just the logon-protocol) is encrypted by an RSA (or an equally powerful asynchronous) algorithm. And on the other hand it is the only technical solution to bypass a firewall without violating the security. Tunneling traffic through SSH requires some expertise in order to create the corresponding background SSH-session, which will furthermore not survive a reboot of the host and therefore may have to be restarted in certain

[10]Spoofi ng [93] has been considered a very unlikely threat.
[11]see http://en.wikipedia.org/wiki/SSH_tunneling for details

intervals. Therefore this mechanism is fully supported by **W2GRID**. The setup as well as the restart is done automatically, once all essential informations have been provided. It requires password-less (public-key) authentication, because **W2GRID** cannot authenticate interactively or supply the password to the SSH-client. The SSH-tunneling also solves the potential spoofing problem, since it is not possible to spoof SSH-tunnelled traffic without prior intrusion into the secure connection. This is unlikely due to the mismatch of effort and potential gain. SSH-tunneling is therefore recommended for a critical environment.

2.3.3 Authentication

All three daemons support user-accounts[12], but since **W2GRID** cannot offer single-sign-on capabilities. The **W2GRID** account(s) should therefore not be confused with Unix/Linux user-accounts on the local or remote hosts. For security reasons it is not recommended to use the same password for both types of accounts.

2.3.4 Logon protocol

In order to connect to a daemon and successfully pass the authentication, the client as well as the server must follow a mandatory protocol before the client can actually send its requests and receive the results. The sequence of queries and answers as illustrated in Fig.2.4 is due to the necessity to get rid of a trouble-making client as quickly as possible.

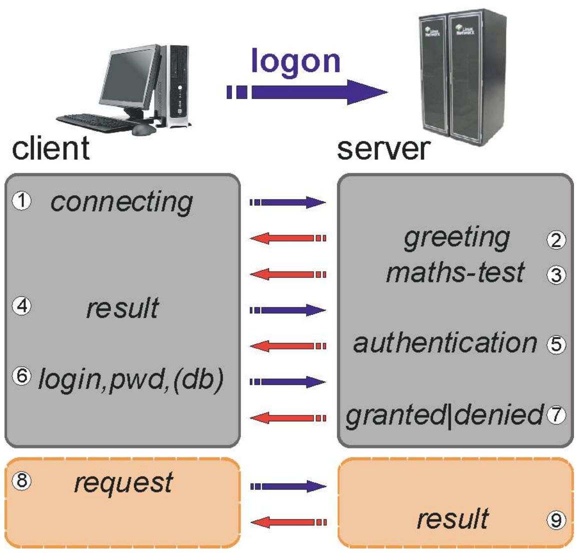


Fig.2.4 illustrates the sequence of requests and replies in the course of a session.

- ***gray boxes:*** *Logon sequence*
- ***orange boxes:*** *Once the client is authenticated, it may submit one or more commands.*
- ***blue arrow:*** *client writes, server reads*
- ***red arrow:*** *server writes, client reads*

The client to the left will initiate the process by establishing a connection to the port, the server (to the right) is listening to.

Figure 2.4: The logon protocol for establishing a connection.

Screenshots of this protocol are provided in Fig.2.13 and Fig.2.29.

[12]While this is an optional feature for the Perl-daemons, it is mandatory for the wiensql-daemon.

1 A client connects to the port, the daemon is waiting on. The client remains silent, while waiting for the server to initiate the process with a short greeting.

2 This greeting string is always sent in plain text and contains the name and version of the daemon. Additionally it may provide informations, if an authentication is required and whether the following traffic will be encrypted or not. The daemon version is foreseen for future development, if the protocol changes for some reason. The version number will ensure, that the appropriate protocol is used by the client. The GridServer and the GridClient provide user-accounts as an additional option, which are disabled by default. The indication whether one of these two daemons will need to authenticate the user is given as a lowercase character prior to the version-number (see Fig.2.29).

3 The server asks the client to solve a simple maths test, which consists of random integer numbers and some of the four different operations ($+$, $-$, %, $*$). This test is already sent encrypted, provided that the encryption has not been disabled. Only if the client possesses the proper encryption-key, it can return the correct result 4 within the few seconds granted before timeout. This approach replaces the authentication for the Grid-Server and the GridClient if the user-accounts are disabled, and on the other hand it renders a brute force decryption much more difficult, since the random numbers will prevent an attacker from using his knowledge about the protocol.

5 If the authentication is required, the server will demand the login and password. Wiensql will additionally ask for the name of the user database (see the screenshot of a wiensql-logon in Fig.2.13).

6 An answer to each query has to be received within a few seconds, otherwise the connection will time out and closed from the server-side.

7 Finally, depending of the success, the server either returns ’access granted’ or ’access denied’. In the latter case it also supplies a reason like ’wrong login or password’ or ’too many clients connected at the moment’.

8 If the client passes the logon, it can continue and submit its requests and receive the results 9. The total number of requests is not limited. If the connection is idle for longer than a given timeout, it will automatically be closed by the server. The client needs to perform the same procedure again for the next request, beginning at 1.

2.3.5 DoS attacks

The basic security measures have already been described above. Apart from AES encryption, SSH-tunneling and the Maths-test, which help to make the brute-force decryption (and hence

the retrieval of the encryption keys) more difficult, the three daemons additionally need to be protected against more dangerous attacks. The DoS attack[13] can be performed with very simple tools and does not even require that the attacker possesses the proper encryption-key. Instead the whole service is simply overloaded by numerous connection attempts. This will at least result in a deadlock of the daemon, hence no other user can connect to it. A more serious attempt however may even crash the host [93].
The DoS attack is aimed at the logon-process. Directly after having accepted a connection from a client $\boxed{1}_{2.4}$, the server immediately forks and continues to listen to its socket for the next connection, whereas its child process meanwhile cares for the authentication of the client. The advantage of this common procedure is, that the daemon can almost immediately serve the next client. Also a crash (due to whatever reason) during the logon will not affect the parent process. The disadvantage however is that every single child process will drain memory and a few CPU cycles from the server. Hence the DoS attack simply tries to open as many simultaneous connections as possible and to send gigabytes of data into the open socket (flooding). As a consequence, the child processes will severely expand, resulting in a memory shortage. Either this or the excess of open sockets will damage the system and eventually force a reboot. Unfortunately this will happen regardless of the permissions granted to the owner of the daemon, hence even a process running under user-permissions can cause the system to fail. To prevent such a situation, several counter measures are applied:

- The total number of connections are limited to fewer than the absolute system-maximum. Once the limit of simultaneous open connections is reached, the daemon will not fork any more unless one of the already open sessions is closed. The logon-process is already cancelled at step $\boxed{2}_{2.4}$.

- To protect against flooding, an open connection must not soak up strings of unlimited lengths from the client, but instead read only a limited number of characters from the input stream at $\boxed{4}_{2.4}$. The first string received by the daemon is supposed to contain the result of the maths-test and never exceeds 32 characters. It is decrypted immediately. If it does not contain the proper result the connection is cancelled and the flooding will be ceased.

- Any service can also be lamed by opening a connection and keeping it open without sending data, which is less critical but annoying. Hence the protocol is deadlocked, as long as the server waits for the result. Since the maximum number of total connections is already limited, this cannot crash the host but it drains all allowed connections. Therefore **W2GRID** defines timeouts for each step of the logon-process. Especially the most critical

[13]Denial-of-Service

reply 4 needs to be sent within 5 seconds.

- These 5 seconds are still enough to lame the service and to prevent authorised users from opening a valid connection (DoS = denial-of-service). This can therefore be overcome either by a shorter timeout or a greater number of allowed connections. None of these choices is appealing. Instead **W2GRID** solves the problem by limiting the total number of connections originating from a single IP to the half of allowed connections. This will prevent a standard DoS-, but cannot cope with a DDos (distributed-DoS) attack, which employs a lot of servers with different IP's.
- Finally it shall be concluded, that an attack may only be averted, if the daemon is able to identify it in due time. Therefore a **W2GRID**-daemon keeps a history of unsuccessful logon attempts. Every time, a new client initiates the logon-sequence, its IP will be checked against the entries of this history. Once a certain client exceeds a given threshold of failed attempts, no further connection from this IP will be served. To keep the number of records manageable, the history is cleaned after a few minutes.

By employing the presented measures, **W2GRID** can counter most DoS attacks but will fail to avert DDos[14] attacks, which are very unlikely anyway due to their effort. Most danger, however can be prevented by using the SSH-tunneling mechanism. The combination of all security measures presented in this section are sufficient to employ **W2GRID** even in critical environments.

2.4 Directory structure

At several occasions later in this text, it will be referred to the location of certain files or executables. For clarity the directory structure is shown and briefly explained here.
The archive of **W2GRID** needs to be expanded into a certain directory (e.g. ♯~/W2GRID/), which is referred to as 'GRIDSRC', because it contains the sources of **W2GRID**. After installation the corresponding path is stored in a variable of the same name ($GRIDSRC). The two additional directories, namely $WIENSQL_ROOT and $GRIDROOT are created during the installation process and afterwards available as an environment variable, too. The actual paths for all three variables can be chosen freely by the user a $_{2.5}$. The $GRIDSRC-directory b $_{2.5}$ contains the unmodified sources and all parts of **W2GRID**, which don't have to be compiled. Also some executables and Perl-tools, which are located in 4 $_{2.5}$. Upon compilation, the binaries and C-libraries are compiled in ♯$GRIDROOT/ 1 and ♯libs/ 2 . Finally the contents of 5

[14]distributed DoS attack: An intruder will fi rst hijack numerous computers and instruct them remotely to conduct DoS attacks. The owners usually don't recognise that their hardware is being abused.

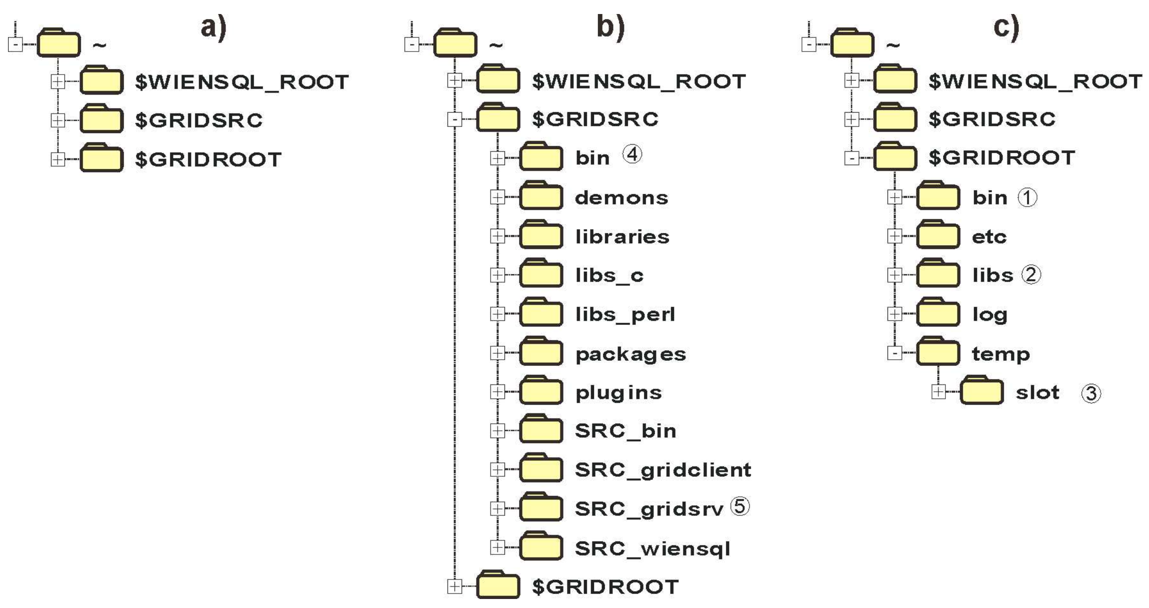


Figure 2.5: Directory structure of **`W2GRID`**

will be given a closer look in section 2.6.2 (see Fig.2.15). Also the temporary directory 'slot' 3 will be explained later (see section 2.6.5.1). Further details are given in the usersguide [97].

2.5 Wiensql-database daemon

W2GRID maintains a very small SQL database for centralised data management. It is developed in accordance with the standalone concept in order to avoid dependencies on third-party software (e.g. mysql or postgre-sql). Whereas commercial or free databases offer much more capabilities, they also require in most cases extended permissions for installation and hence erode the independence of the infrastructure. Usually there is one wiensql-daemon per user and host, which can serve any or both of the Perl-daemons illustrated in Fig.2.1.

2.5.1 Purpose

The Perl-daemons and several tools have to access and manipulate a lot of data at run-time, which may be written, read and updated by different concurrent processes. To avoid data-corruption, these processes need a proper data management, which is best served by a database daemon. Consequently it will serve as the only authority, which is allowed to read and write directly from and to the respective files, whereas its clients have to manipulate the data in an abstract way. The constraints enforced by the use of a database provides data-consistency, since all instructions for data-manipulation and data-retrieval have to conform to the syntax of the employed language (SQL[15]). The following list shows a few practical examples, where the database comes to play.

- The registry (see section 2.6.6) contains the settings of the daemons ('hostinfo.server' or 'hostinfo.client') such as the IP, the port, the hostname or the encryption-key. These informations are needed for the start-up as well as for the logon-process to authenticate a client.

- if the authentication is enabled the daemons need to maintain user-accounts in order to compare the transmitted login and password of a connecting client with the authorised entries.

- When a calculation is started on a GridServer, it will be associated with a certain PID[16]. In order to check whether the calculation is still running or already done, the GridServer must use this PID to retrieve the corresponding informations from the underlying execution-layer (e.g. PBS). The script, which is responsible for performing this check is executed in regular intervals (see jobs in section 2.6.3). The PID and a lot of additional data have to be stored intermediately, otherwise they could not be recovered with the next check. (see slots in section 2.6.5.1)

[15]Structured-Query-Language
[16]This can be any string or number

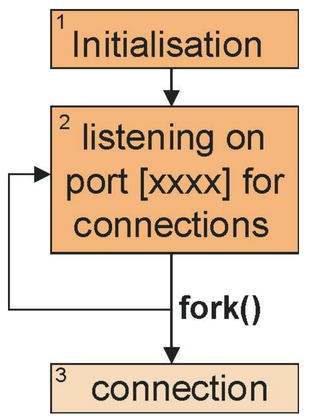
- **`W2GRID`** performs regular tasks such as the check of a calculation or the check of the slots. The Perl-daemons has to be told, which tasks are to be executed and when. Such informations are supplied from a table (see job-registry in section 2.6.6)
- When searching for resources to submit a calculation, the GridClient needs to know a description of is GridServers, the method of how to connect to them and how to transfer data. Thus it requires the knowledge of some static informations like the total-memory or the available applications (see the host-registry in section 2.6.6).

2.5.2 Workflow

The wiensql-database daemon must provide an excellent performance and avoid to consume excessive memory otherwise it will be a bottleneck for all other processes. Therefore it was implemented entirely in 'C'. In order to be portable to any architecture it uses only standard libraries, which come by default with every compiler installation. Its simple workflow is shown in Fig.2.7. At startup the daemon reads its connection-parameters and some additional data such as security settings from the file .wiensqlrc† (Fig.2.6), which is also read by its many wiensql-clients (lines 1 - 4) to determine, which port the daemon is listening on.

```
1: server:athena
2: port:18847
3: key:YnZpe5e4w0Nd
4: database:athena
5: max:30
6: allow:*.*.*.*
7: timeout:120
8: cmdline_posix:1
```

Figure 2.6: Sample content of the wiensql startup-fi le .wiensqlrc†



1 *wiensql reads a startup-file (.wiensqlrc†) and extracts the port (line 2) and the encryption-key (line 3). The lines 5 - 7 affect the security policy.*

2 *The daemon binds to its assigned port and listens for incoming connections.*

3 *Child processes serve the clients.*

The parent will quit the loop if it receives a proper external (SIGINT,SIGTERM,SIGKILL) or internal signal (SIGUSR1).

Figure 2.7: Workfbw of the wiensql-daemon

2.5.3 Accessing the database

Only the wiensql-daemon is allowed read and write directly from and to the database-tables (i.e. the respective files). Other processes (programs) have to create a tcp-connection to the wiensql-daemon and manipulate or request the data by the use of SQL strings. These client-functionalities are at present only implemented in Perl as a single library libs_perl/wiensqllib.pl[†]. Providing a C-implementation of these functions has not been necessary so far, and the C-Shell lacks sufficient capabilities.

2.5.3.1 Commandline interface

The user does not need to access the database interactively. For debugging purposes, **W2GRID** provides a commandline interface '***wiensql.pl***', which also supports batch mode. Its usage will be described in section 3.2.3.3.

2.5.3.2 Perl-tools

Several executables, especially those used for the installation (or a later modification) of **W2GRID** need wiensql-support. Fig.2.8 shows an example, how the respective library is included and used from within a Perl-program. In contrast to the daemons the tools will usually not fork, and hence not establish more than a single connection, which is therefore easier to handle. The contents of the startup-file are read in line 2, since the client needs to know the

```
1:  require "$PATH{LIB}/wiensqllib.pl";
2:  &readwiensqlrc();
3:  &wiensql__setlogin("gridclient_athena");
4:  if(!&wiensql__connect())
5:  {
6:      &handle_extra_output();
7:      exit(1);
8:  }
9:  my $display=0;                                #(tabular output)
10: my $request="select * from hostinfo.client";  #some command
11: my $mode="sql";                               #treats $request as SQL-string
12: my ($result,$errors,$warning)=&wiensql__exec($request,$display,$mode);
13: print "$result\n";
```

Figure 2.8: A Perl-code fragment to demonstrate the use of the wiensql-client library

encryption-key and the port. (Fig.2.6). If the authentication succeeds (line 4), the client can submit the request (line 10), and will receive the result (line 12). The output is available in different styles (line 9), among which the tabular format is the most common one. Additionally the daemons knows an “sql” mode and a “cmd” mode for non-SQL strings (line 11).

2.5.3.3 Perl-daemons

Different to the Perl-tools, the daemons will fork a couple of times and also create and maintain several simultaneous connections, which is the final reason for the demand to obtain a database with low memory consumption. A wiensql-connection should not be shared between a parent and its child process, since the behaviour of an open tcp-socket is unforeseeable if one of the involved processes is terminated. Therefore the wiensql-connection is closed in the child process and reopened, which results in several independent connections being served at the same time. These are especially numerous in the case of a parallel workflow as illustrated in Fig.2.9, where each 'DB' icon in the drawing represents an individual connection. The figure already anticipates the workflow of a command, which is explained in detail in Fig.2.14 and Fig.2.17. Every child process of a Perl-daemon (such as the GridServer), 1 $_{2.9}$ needs its own

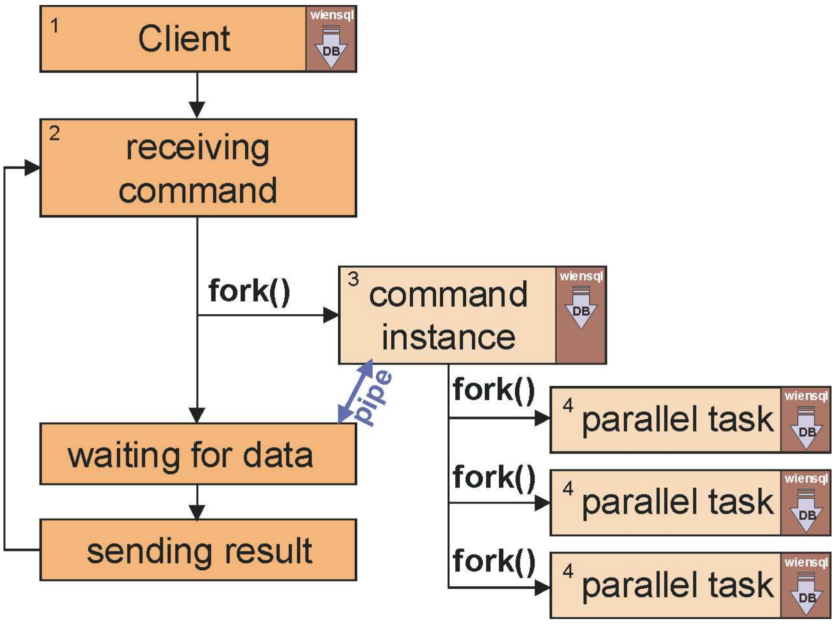


Figure 2.9: Abstract view on the number of independent database client sessions, that are created throughout the workflow of a command

database-connection to retrieve login and password in order to authenticate the client. Once successfully logged on, the client submits requests 2, which is processed by another child (grandchild) 3. If the included workflow contains parallel tasks like querying GridServers, each of these 4 might need an additional database-access[17].

[17]e.g. for retrieving the connection parameters of each individual host

2.5.3.4 C-Shell scripts

Numerous inserts and updates have to be done in the course the installation of **W2GRID**. Since the C-Shell is not capable of connecting directly to the wiensql-database, the script employs the commandline interface ***wiensql.pl*** in batch mode. A sample script is provided in Fig.2.10. Yet if the performed operations are more complex and require several subsequent

```
#!/bin/csh
set sql = "select * from hostinfo.client"
set login = "-l_sql gridclient_athena"
set result = ‘wiensql.pl $login -S "$sql"‘
echo $result
```

Figure 2.10: Sample code, which illustrates the use of the wiensql-database from within C-Shell scripts

operations and intermediate regular expression manipulation, the whole database-workflow is coded entirely in Perl and provided as a Perl-tool. This tool is invoked from within the csh-code with proper arguments. Such a tool is provided for example for installing the proper platform plugin (see Fig.2.11). The argument to be supplied (line 5) is the ID of the plugin (see section 2.9), which is selected from a list (line 4), supplied from another Perl-tool in line 2. The code-snippet does not show any error-checks.

```
1: #!/bin/csh
2: listplatforms.pl
3: echo -n "     select the platform: "
4: set platform = $<
5: install_platform.pl $platform
```

Figure 2.11: Complex database-operations in C-Shell script are better accomplished by the use of Perl tools.

2.5.4 Optimisation

According to the illustration in Fig.2.9 numerous connections may be open simultaneously. As a matter of implementation, even a simple RPC command submitted to one of the Perl-daemons (Fig.2.1) will require at least two connections, and each of them performs a full logon-process starting from $\boxed{1}_{2.4}$. To avoid a bottleneck, the protocol must be simplified, which is possible by reducing the number of strings being sent across the network as illustrated in Fig.2.12. The potential for the optimisation comes from the delay of the socket-write operation and from the encryption of each individual string. Reducing the number of these operations significantly improves the performance. The maths-test, login, password and database queries are not performed sequentially as illustrated in $\boxed{a}_{2.12}$, instead the client will pack everything into a single string, thus performing only a single encryption and a single write-operation $\boxed{1}$.

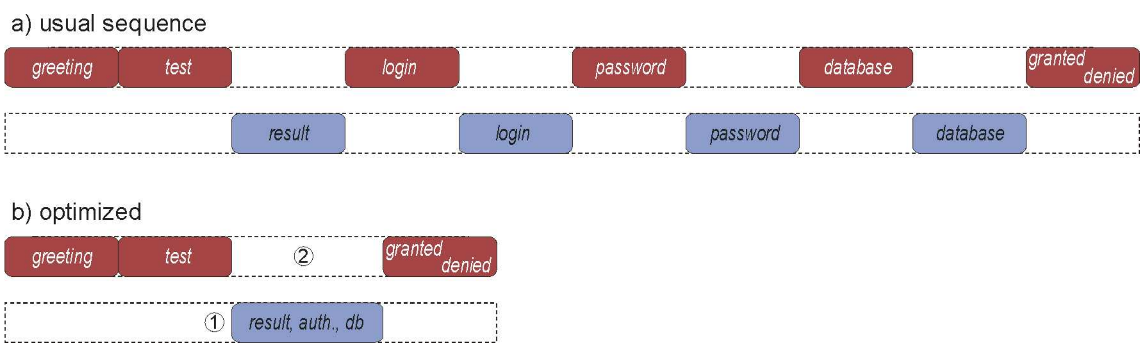


Figure 2.12: Optimisation of the logon-process

The server decrypts the string and extracts the required results for all the individual requests 2. If any of the results is missing in the string, it will be requested separately as usual[18]. The optimised process b comes with only three read/write and encrypt/decrypt operations instead of the nine of before. A screenshot ofthe verbose output, obtained with the improved logon protocol is shown Fig.2.13.

```
[SOCKET] Trying to connect to athena on port 18837!
[SOCKET] Socket creation succeeded
[SOCKET] Bind succeeded
[SOCKET] Connect succeeded
[SOCKET] reading 'WIENSQLD v:3.0'
[INTERNAL] connection to wiensql demon version '3'
[SOCKET] reading '********************'
[INTERNAL] READING ENCRYPTED 'tellme:690448+54564852+82068889+18622724'
[INTERNAL] we are connected to a server, which can read the complete greeting
[INTERNAL] SENDING ENCRYPTED: '155946913;login=gridsrv_athena;pwd=abc;db=athena'
[SOCKET] sending '****************'
[SOCKET] reading '*************'
[INTERNAL] SERVER said:access granted
```

Figure 2.13: Verbose output of ***wiensql.pl*** collected during a connection attempt to the wiensql-daemon

2.5.5 Security measures

The principal step is to keep the encryption-key secret. Since the data is stored in a file located in the ♯$WIENSQL_ROOT/-directory, it is by default not readable by anyone but the user.
To avert random attacks, the wiensql-daemon implements the security mechanisms, which have been explained in section 2.3.5. The total number of connections is limited to a maximum of 30 by default, but this setting can be changed in the file .wiensqlrc† (line 5 of Fig.2.6). Additionally the IP of all clients, which have successfully logged on is stored in a shared memory segment. Once the maximum number of clients is reached, the daemon will not accept

[18]The sequence of the individual requests (login, password, database) is not mandatory anyway.

any further connection until one of the existing ones is terminated. Idle connections are closed automatically by the server after a certain time of inactivity (line 6 of Fig.2.6). It is also possible to constrain the IP of the client to e.g. the IP of the local host. The example shown in Fig.2.6 (line 6) does not instruct the daemon to impose any such restrictions (*.*.*.*).

2.5.6 Supported data-types

A definition of the available wiensql data-types is presented in table 2.1.

name	***corresponding C-type***	***purpose***
tinyint	char	small INTEGER
mediumint	signed short int	medium INTEGER
int	signed int	regular INTEGER
bigint	signed long int	big INTEGER
char	char (1-255)	string (fixed length)
varchar	char (1-255)	string (variable length)
float	double	the only floating-point type
time	time_t	HH:MM:SS
date	time_t	dd-mm-yyyy
datetime	time_t	HH:MM:SS dd-mm-yyyy
ipaddr	char (6)	IPv4 or IPv6
tinytext	char (1-8192)	regular string
mediumtext	char (1-32768)	medium size string
text	char (1-131072)	long string
bigtext	char (1-1048576)	very long string
encrypted	char (1-255)	encrypted in the table-file

Table 2.1: Wiensql data-types

2.6 GridServer and GridClient daemons

The two Perl-daemons 2 $_{2.1}$ and 5-7 $_{2.1}$ serve different purposes (see section 2.2) but the overall principle and the methods of operation are identical and can be illustrated by the simplified scheme in Fig.2.14. The daemons must accept and verify incoming connections and respond to the requests by executing RPC-workflows (see section 2.6.2), but also run scheduled tasks in regular intervals (see section 2.6.3). Both tasks are served in an efficient way by two different processes.

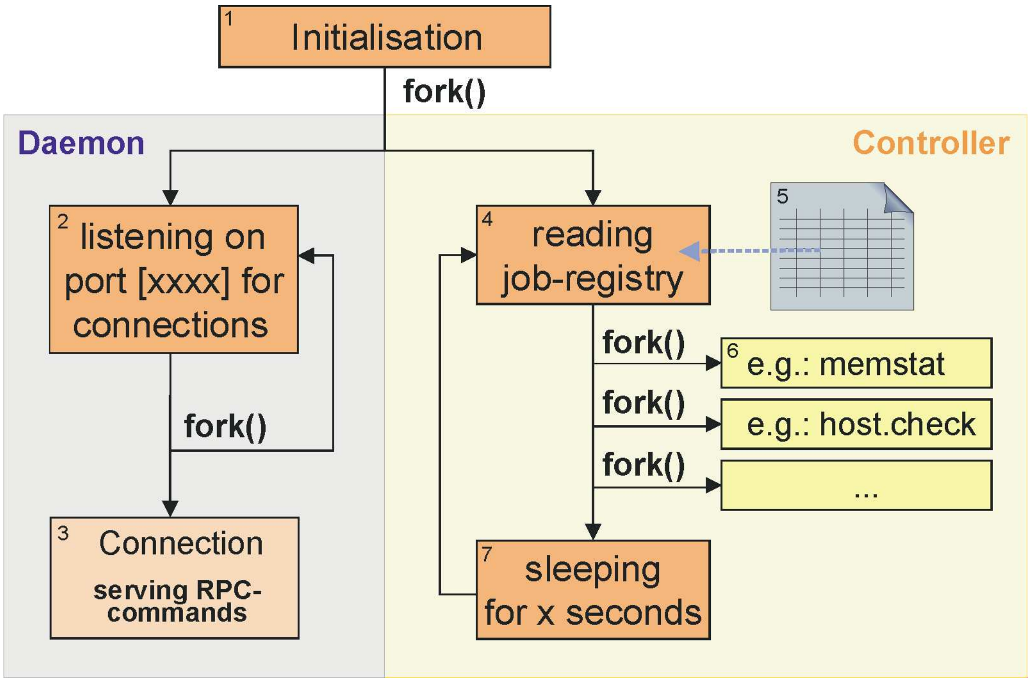


Figure 2.14: Workfbw of a **`W2GRID`** Perl-daemon

1 As a pre-requirement, the wiensql-daemon must be online and operable, thus each Perl-daemon needs an account and an array of tables with proper data being filled in (see section 2.6.6). If this is provided, the process will fork[19], since the diverse tasks (see above) can best be served with two independent threads.

2 One instance of the original process binds to the specified port and waits for incoming connections (e.g. from a commandline interface or the GridClient daemon). This instance is the actual **daemon process**. Each client[20] causes this process to fork again.

[19]Forking means to double an existing process, generating an identical copy in memory, which has a different PID. The original process is called the 'parent', whereas the newly created one is the 'child'

[20]The incoming connection is always referred to as the 'client' [89], which must not be confused with the GridClient, whereas the listening process is called the 'server', which is again not to be confused with the GridServer!

This is essential, otherwise the daemon could not serve the next client.

3 This child process will perform the authentication (see section 2.3.4), execute all the incoming requests and return the results to the client. The illustrated instance is shown in further detail in Fig.2.17. The process is sequential, hence no further request can be read and processed unless the current one is done. The individual requests are run in 'foreground' in contrast to 6. When the connection is closed either by the client on purpose or by the server due to an exceeded timeout, this instance will be terminated.

4 The second process created after initialisation is used for executing regular tasks such as checking running calculations or the cleaning up of temporary-directories. The purpose of these regular tasks is comparable to 'cron jobs' on Unix, details will be given in section 2.6.3. This process, which controls these regular tasks and serves a similar purpose as the 'cron daemon' is correspondingly called the **controller**.

5 A database-table - the job-registry - maintains the scheduled tasks. The controller 4 reads the table in regular intervals. This interval is by default set to sixty seconds, which is sufficient for all **`W2GRID`**-tasks (see the examples on page 57). For this reason, the jobs cannot be executed exactly at a given time (e.g. in contrast to real cron jobs), instead the execution-date given in the table just refers to the time, until which the job is delayed. Once this date is exceeded, it will be run by the controller in its next check-cycle.

6 All the scheduled tasks are processed in parallel in the background in contrast to the commands 3, hence the controller has to fork as many times as it has got entries in the table, whose execution date is already exceeded. Each one is an independent process. It is left to the task to remove or update its entry in the table or even leave it unchanged. In the latter case it will be run again with the next check-cycle, since the date remains in the past.

7 The controller does not wait for the jobs 6 to be finished. It cleans up and remains idle for the given interval between each check-cycle. This idle-phase will use the 'sleep()' command, which causes the process to stop execution unless it receives a TIMER signal from the operating system.

2.6.1 Directory structure

To improve the readability of the explanations given in some of the following subsections, it will be referred to the contents of the respective daemon directory, which is shown in Fig.2.15. Since the directory structure of both Perl-daemons (♯SRC_gridsrv/ and ♯SRC_gridclient/) is identical, only one is shown here. For further informations it is referred to the usersguide [97].

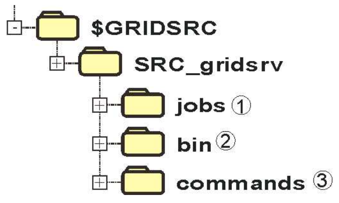


Figure 2.15: Directory structure of ♯SRC_gridsrv/

1 Contains the Perl-scripts of the background processes 6 $_{2.14}$ (section 2.6.3) and may also have subdirectories.

2 Some (Perl-) tools are required for the installation like ***listplatforms.pl*** or ***install_platform.pl*** as shown in screenshot Fig.2.11.

3 Contains the Perl-scripts of the foreground processes 3 $_{2.14}$ (section 2.6.2) and may also have subdirectories.

2.6.2 Commands (foreground tasks)

Both Perl-demons export their capabilities as RPC-commands (or briefly 'commands'). In order to improve the reading of this paper, the commands are enclosed in curly braces and labelled with a corresponding subscript-character, indicating which daemon offers it. A Gridclient-**{command}**$_C$ will have a subscript 'C' attached to the command-name, whereas a GridServer **{command}**$_S$ gets a subscript 'S'. Several **{commands}**$_{S,C}$ are available by name on both daemons, the GridServer and the GridClient as well, but the respective source code will be different. Those commands have both characters attached to their command-names.
The daemons accept such requests only by ways of an authorised tcp-connection 3 $_{2.14}$, which can be established by any client, such as the commandline interface. Its behaviour is similar to any standard Unix/Linux terminal session with the respective usability.
The command, which is actually a workflow provided by a Perl-script can be followed by one or more mandatory or optional arguments. It is included at runtime into a child process of the daemon code. The naming convention, applied by **W2GRID** will rule, that the command-name is also the filename (rump without the Perl-extension .pl).

Example:

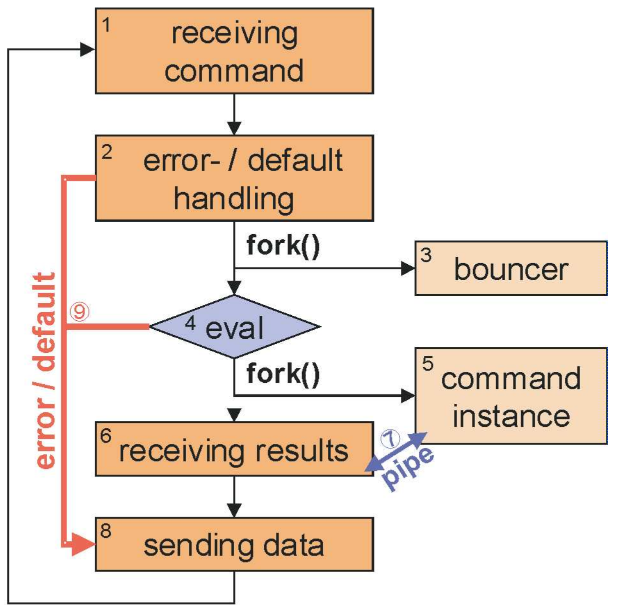
The command **{test}**$_{S,C}$ *will return a simple output (something like 'Hello World'). It is invoked by typing the string 'test' into the commandline interface, which is subsequently sent to the daemon. The file* test.pl† *contains the respective workflow, whose source code can be found as an example in the appendix (see Fig.A.4). The screenshot of its output is shown in in (Fig.2.16)*

Each command triggers a complex workflow, which is governed by the necessity to provide

```
>test
#command took 0 seconds to complete
If you read this the gridserver is operational
>
```

Figure 2.16: Sample output of the command **{test}**$_S$

certain failsafe mechanisms as illustrated in Fig.2.17.



The content of this figure is a more detailed view on the processes, which are triggered after a connection has been established by a client 3$_{2.14}$.

- ***dark orange box:*** *parent process.*
- ***light orange box:*** *child process.*
- ***blue arrow:*** *pipe for interprocess communication between parent and child.*

The additional processes serve for failsafe and to enhance the stability. Further it helps with the debugging, since feedback on errors, which occurred in the command-script can be caught and sent to the client.

Figure 2.17: The workflow triggered in respond to an incoming command

The processing is sequential, only a single command can be executed at a time. The connection will not process any further input unless the last one is finished.

1 A command is received as a string and split into two pieces at the first blank, of which the first one has to contain the command-name, whereas the second (optional) piece is supposed to contain additional arguments. This workflow-element is part of 3$_{2.14}$.

2 Some 'built-in' or 'default' -commands bypass the complicated workflow and do neither

need the Bouncer-process 3 nor require to be processed by a separate child 5 and hence return the result much faster. These are: **{?}**$_{S,C}$ **{help}**$_{S,C}$ **{whoami}**$_{S,C}$ **{ping}**$_{S,C}$ **{version}**$_{S,C}$ **{exit}**$_{S,C}$ **{quit}**$_{S,C}$ **{shutdown}**$_{S,C}$.
If the process encounters a non-fatal error (e.g. no wiensql-database connection possible) at this state, it will bypass the usual processing too and return the error-message to the client.

3 The first child instance, which is created by the daemon is the so called 'Bouncer', which serves two very important purposes and is explained in detail in section 2.6.10. On the one hand it keeps a connection open, which would otherwise be closed after a certain time of inactivity. This is important if the workflow of the triggered command takes more than the given timeout to be processed. And on the other hand its periodically transmitted data-packets are hitchhiked by a mechanism, which forwards informations about the progress of the current command-processing. The Bouncer has to be terminated before the result is returned to the client 8.

4 All non-default[21] commands have to exist as a Perl-file in ♯SRC_gridsrv/commands/ 3$_{2.15}$. If this is not the case, the client will receive an error-message. A command must not contain a dot • in its name (see 2.6.2.2) and needs to be spelled correctly, since the daemon is case sensitive.

5 The actual command-file is not directly processed by the parent 1 for failsafe. If the included Perl-file contains an error, the whole process would be terminated and there is no instance left, which could report the reason for the error to the client. Therefore the parent forks again and delegates the execution of the command to its child, which includes the respective file and calls the entry-function of the workflow (&exec_request()). A sample code is provided in the appendix in Fig.A.4 (see line 12). In the case of a fatal error, only this child process will crash, but may still forward the error-messages to the parent 6. Non-fatal errors (e.g. invalid input) can be handled by the command itself and will be reported like a standard-result over the pipe 7.

8 The parent must safely terminate all its child processes 3 and 5 before it sends the result-data to the client and returns to waiting for the next command.

[21]Those, which are not built-in and do not bypass the regular processing.

2.6.2.1 RPC-stubs

RPC-commands are often referred to as 'RPC stubs', because they complement each other. and contribute to a bigger workflow, which extends the scope of each single one. In the illustrated example (Fig.2.3), there is an RPC stub called **{load}**$_C$, which is requested by the user. This in turn invokes the corresponding **{load}**$_S$ stub of the GridServer. The user will receive only the final result $\boxed{4}_{2.3}$ and not the intermediate ones. Not everything is implemented in a stub-like manner. There exist certain (simple) commands (e.g.**{help}**$_{S,C}$), which do not depend on others.

2.6.2.2 Grouped commands

Having to work with numerous individual commands is found to be quite inconvenient and difficult to overlook. Furthermore each command-name could only be used once, since they would all be located in the same directory. To overcome these constraints **`W2GRID`** allows to 'group' commands (see section 2.6.3). The grouping is directory-based, hence each subdirectory of $\boxed{3}_{2.15}$ is the group-name and the files contained therein belong to the group. To separate the group- and the command name, **`W2GRID`** uses a dot.

Example:
{host.list}$_C$ *shows all registered GridServers and their properties, which are known to the Grid-Client. The string 'host' is the name of the group (i.e. the subdirectory name), and 'list' is the name of the command*[22]

As a result of this approach there may be several commands having the same command-name, provided they belong to different groups and can be distinguished from one another.

Example:
{host.list}$_C$ *vs.* **{job.list}**$_C$. *The commands are not identical although both filenames are 'list.pl', because they are located in different directories.*

2.6.3 Jobs (background tasks)

The infrastructure makes use of scheduled workflows, which are executed in the background in regular intervals. Since they can be compared to 'cron jobs' they will be simply called 'jobs' in

[22]The workfbw can be found in the fi le $GRIDSRC/SRC_gridsrv/commands/host/list.pl[†]

this text. As this term has intuitively a different meaning in common speech, the **W2GRID** background $\boxed{JOB}$ is marked with rectangles wherever it appears to be ambiguous. The reading is further improved by a naming convention similar to the commands. Each $\boxed{job}_{S,C}$ is enclosed in rectangles with a subscript character attached, which refers to the respective Perl-daemon ('C' for the GridClient, 'S' for the GridServer).

Since **W2GRID** only wraps applications and does not use an invasive approach like other middleware, it lacks the ability to react immediately to events (e.g. completion of a calculation). Instead it has to check in regular intervals if a certain observable property changes (e.g. the PID has vanished from the process-table, indicating that the calculation is finished). Hence in contrast to other approaches, **W2GRID** does not get notified by the system or the application, and has to capture the event by its own means. In most cases this approach means a delay of roughly one minute to a workflow, which is usually not significant in comparison to the overall runtime of realistic tasks.
The procedure is shown in Fig.2.14. The controller $\boxed{4}_{2.14}$ will iteratively check its schedule $\boxed{5}_{2.14}$ and execute those $\boxed{JOBS}$ with expired target-dates $\boxed{6}_{2.14}$. The purpose of this mechanism shall be explained by the use of some examples:

- The GridClient keeps a list of GridServers in its host-registry. The job $\boxed{host.check}_{C}$ contacts the daemons regularly and reports broken links to the user. Additionally it will request and compare the static host-informations with the entries stored in the local registry, and update the same if necessary [98].

- On unmanaged desktops, the GridServer needs to keep record of the overall memory utilisation to avoid causing memory bottleneck. For this purpose, the job $\boxed{memstat}_{S}$ will be run in intervals of roughly two minutes, fetch the most recent memory statistics and write it to a database-table.

- The two examples above remind of the purpose of cron jobs, which is undeniably true. Yet a more important feature of the $\boxed{JOBS}$ is the fact, that they are run in the background. There are ways to exploit this in order to make workflows becoming smoother. Tasks like the filetransfer for example can take a long time to be completed. If such a time-consuming task is performed by a command-script, the commandline will be blocked for the whole time it takes to complete the copying. The user, who submitted the command will have to wait for the filetransfer to be completed until he can submit the next one. This is due to the blocking fashion [99] of RPC's, but such a task can better be accomplished in the background, by generating the respective $\boxed{JOB}$ from within a command and starting it immediately. While the $\boxed{JOB}$ performs e.g. the filetransfer, the user may submit another command. Since the output of this background process cannot be sent to the socket (the connection stays with foreground processes), the $\ulcorner JOB \urcorner$ must write it to a database-table. The principle of this process is illustrated in Fig.2.19).

A sample Perl-script of such a job (e.g. $\ulcorner test \urcorner_S$) is presented in the appendix (Fig.A.5). The scripts are located in directory $\boxed{1}_{2.15}$ or in one of its custom subdirectories.
The $\ulcorner JOBS \urcorner$ add a lot of flexibility to the middleware and improve the overall performance significantly. For writing efficient workflows (e.g. the submission of a calculation) both, commands as well as $\ulcorner JOBS \urcorner$ are required (see section 2.6.4).

2.6.3.1 Grouped jobs

The same feature, which turns the numerous commands into an organised list can be applied for the $\ulcorner JOBS \urcorner$.

2.6.3.2 The job-registry

The database-table $\boxed{5}_{2.14}$, which contains the schedule of the regular background tasks has already been mentioned earlier. Each of the two Perl-daemons owns such a table, whose definition is shown in Tab.2.2. Only the essential columns are represented, since the definitions on both daemons are not completely identical. The differences, however, are not essential for explaining the principle. A unique integer **(job_id)** is used to identify the $\ulcorner JOB \urcorner$ distinctly. Several commands, whose purpose is to manipulate certain entries of this table (e.g. **{job.kill}**$_{S,C}$) will need it as a mandatory argument, in order to retrieve the respective row in the record-set. Additionally, the $\ulcorner JOB \urcorner$ has to be linked to the corresponding Perl-script, which contains its workflow and can be found either directly in directory $\boxed{1}_{2.15}$ or in one of its subdirectories. This information is container in the column **(command)**. The **(job_date)** finally defines the time, when a $\ulcorner JOB \urcorner$ is to be processed. For clarity it must be stressed again, that the controller cannot run a job at exactly that time, which is specified in the registry, because it will check the re-

column	*datatype*
job_id	int
command	char 50
state	int
job_date	datetime
pid	int
parameter	text

Table 2.2: Essential columns of the job-registry

spective table only in regular intervals. For that reason, the job-registry should not be confused with a schedule like those of the cron daemon. Instead the controller fetches all entries, whose corresponding **(job_date)** has already passed. All $\boxed{JOBS}$ in contrast, whose **(job_date)** is still in the future are not affected. In order to execute the code, the controller forks and generates a child process, which will subsequently include the corresponding Perl-file and invoke its entry-function (line 9 of Fig.A.5). The PID of this child will be stored in the table in column **(pid)** for failsafe, because as long as this PID is recognised by the operating system, the $\boxed{JOB}$ is considered to be running and will not be started again by the controller, even it the **(job_date)** may still be in the past. This prevents a harmful racing condition, such that several instances of the same $\boxed{JOB}$ are running in parallel. Before the script exits, it has got to update its own entry, either by updating the **(job_date)** to a time in the future (line 28) in order to run again or by deleting the row from the table (line 34). If for instance the $\boxed{JOB}$ quits without updating the table in any of the two possible ways, it will be run again in the next cycle, because the row remains in the table with a date in the past. The **(state)** of a $\boxed{JOB}$ offers the feature to subdivide the workflow into sections. The first section 'init' is executed with the first run of the script. If the **(state)** is set to e.g. 'running' afterwards, the other section -provided it exists in the workflow- will be execute with the second run, whereas the 'init' part is skipped. A $\boxed{JOB}$ may additionally contain some custom data, which are stored in **(parameter)**. As a matter of convenience these are mostly XML-formatted strings, as shown in the example of Fig.2.18. The interested reader is referred to the appendix (page 163) for the purpose of the Perl-functions used in the code. . The lines 1 - 5 define the contents of **(parameter)**, which will be the following XML-

```
1: my %jobdata=&new_dtgr("PARAMETER");
2: &dtgr__additem(\%jobdata,"A","",1);
3: &dtgr__additem(\%jobdata,"B","",2);
4: &dtgr__additem(\%jobdata,"C","",3);
5: my $parameter=&dtgr__data2str(\%jobdata);
6: my $command="wien.exec";
7: my $job_date="20s"; #will be stored as now+20 seconds
8: my $local_job_id=&jobutils__newjob($command,$job_date,$parameter);
```

Figure 2.18: Sample code snippet, which inserts a new item into the job-registry

string: "<PARAMETER><A>1</A><B>2</B><C>3</C></PARAMETER>" line 8 finally inserts the $\boxed{JOB}$ into the job-registry and returns the **(job_id)**.

2.6.4 Interplay of commands and jobs

It has already been mentioned, that for realistic workflows such as the execution of applications commands and $\boxed{JOBS}$, whose purpose and concept have already been discussed intensively above, are required. In this section it shall be demonstrated in simple terms, how these two elemental building blocks can be employed to perform a complex workflow, which is composed of a filetransfer, the subsequent calculation-start and frequent monitoring of the same until the calculation is done. In principal one could use a single command, which contains all these tasks, however there are two disadvantages:

- The design of the RPC-commands (see Fig.2.17) does not allow that any result is sent to the client unless the whole workflow has been processed (see line 39 of Fig.A.4). The connection will be blocked and no other command can be sent until this line is reached. That means, that the commandline is not usable for as long as the process takes to be run in the foreground, which is in this case the duration of the whole calculation (maybe hours).
- Yet more cumbersome, the execution of a command will be stopped if the server looses the connection to the client, which may happen either accidentally or on purpose. In any case it is especially troublesome, if the command takes several minutes or even hours to complete, because all data is lost.

An appealing feature of **W2GRID** is, that the middleware leaves the implementation of the workflows completely to the developers and provides only a large set of tools and libraries, hence it is up to the person who implements the workflow, whether to use the unfavourable solution with a single command or to employ $\ulcorner JOBS \lrcorner$ as illustrated in Fig.2.19 to profit from the flexibility of background processes. The command-names **{optimal.exec}**$_C$ and **{optimal.info}**$_C$

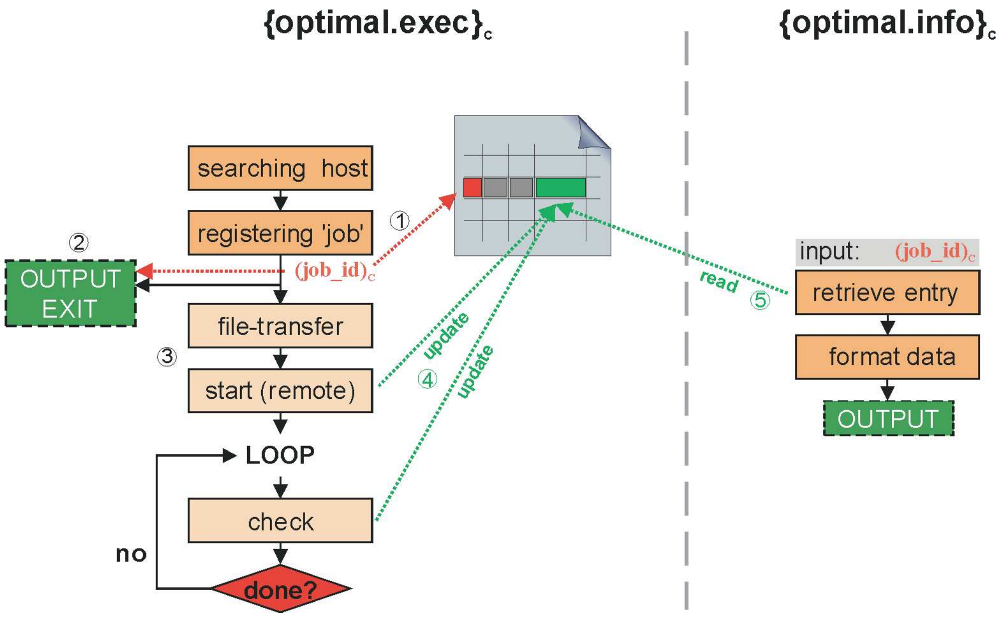


Figure 2.19: An example for the interplay of **{COMMANDS}** and $\ulcorner JOBS \lrcorner$

in this example are fictitious and serve only for a demonstration of the principle of interplay.

The left-hand side of Fig.2.19 shows the initial command, which has been submitted to the daemon. It starts the workflow by registering the $\ulcorner JOB \urcorner$ $\boxed{1}$ and obtains the associated unique **(job_id)**, which is returned as a result to the client $\boxed{2}$. Before the command quits, it invokes the $\ulcorner JOB \urcorner$ $\boxed{3}$, which runs in the background and is disconnected from the socket-connection. All the client has got is the **(job_id)**, which ultimately refers to the corresponding entry in the database. Since the output cannot be sent to the socket any more, the $\ulcorner JOB \urcorner$ must write important data such as the progress of the calculation into the respective column **(parameter)** $\boxed{4}$, and the same data may be retrieved $\boxed{5}$ by the aid of the command **{optimal.info}**$_C$ at any time. This command requires the **(job_id)** and simply reads the data, which is constantly updated in the background. If the $\ulcorner JOB \urcorner$ is finally done, the respective entry will be removed from the table, and the **(job_id)** is not recognised any more, which is similar to the behaviour of common Unix/Linux background processes. Hence the job-registry can be qualified as a non-persistent data container, because the lifetime of the respective entry expires with the lifetime of the job. This is okay for most purposes, otherwise a persistent data container must be employed, which will be explained in the next section.

2.6.5 Persistent data containers: 'Slots'

Sometimes **W2GRID** requires a data-deposit, which outlasts the scope of a $\ulcorner JOB \urcorner$ and allows to store informations for later use. This is however not their only purpose. The illustrated example in Fig.2.19 was deliberately set up for the GridClient, because its files are supposed to be 'local'. The situation is different for the GridServer. Most applications usually need some input, and if the files of the GridClient are not accessible, they have to be transferred prior to starting the calculation. But where are they copied to? This is the first and predominant purpose of a slot: To distinctly identify a certain (temporary) directory[23] -whose absolute path does not have to be known to the client- by a unique ID, which is called the **(slot_id)**. A convenient set of commands allow to create, delete and manipulate the slots by a given **(slot_id)**.
Because the slot is just an entry to a table (the slot-registry), it can be linked with any kind of data. This is the second important feature, which shall be explained in detail. In contrast to the data stored in the job-registry, which is perished after the $\ulcorner JOB \urcorner$ has come to an end, the slot-data are persistent and have to be deleted on purpose by the aid of a certain command **{slot.kill}**$_C$ or **{slot.release}**$_S$. This property provides a lot of additional features for complex workflows:

- It can serve as a 'history' for calculations.

[23]It is just temporary on the GridServer. The GridClient in the opposite requires only a pointer to the local directory.

- The final state of the calculation is conserved, whereas a ⌜*JOB*⌟ will have to transfer all its content to a file, otherwise it is lost.
- Numerous ⌜*JOBS*⌟ can make use of the same slot, either at the same time or sequentially (e.g. the filetransfer and the monitoring of a calculation could be two independent ⌜*JOBS*⌟)
- The filetransfer can make use of the 'abstract' ID, which is more convenient than working with absolute pathnames.
- All relevant informations and methods can be pooled and accessed with a single ID. This **(slot_id)** can be re-used (e.g. the same calculation can be rerun), whereas the **(job_id)** is deleted when a ⌜*JOB*⌟ expires.

The advantages of the slot are quite pervasive, therefore it should be used by default for every bigger workflow, which employs elements such as filetransfer and remote operations in a similar way as illustrated by Fig.2.20. In this example, the **(slot_id)**$_c$ uses the attached char-

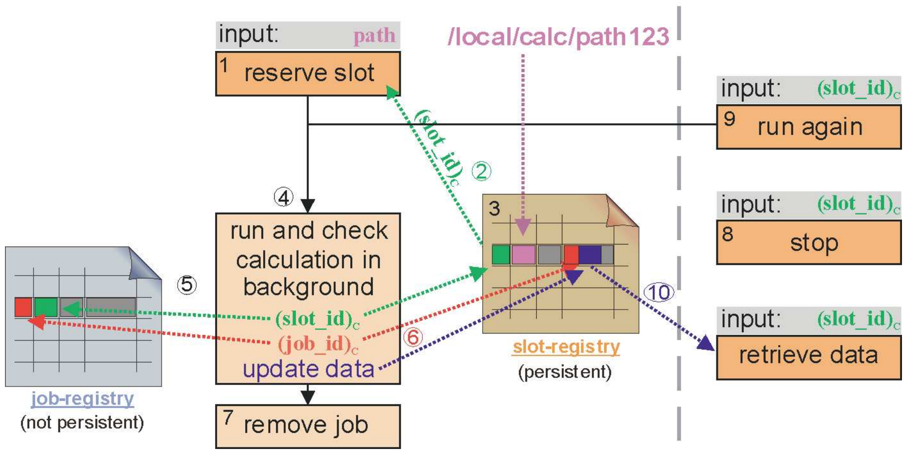


Figure 2.20: The purpose of a slot as a persistent data container.

acter 'C' to indicate, that this ID is supposed to identify a slot of the GridClient. Common workflows usually start with the slot-reservation $\boxed{1}$, which returns the corresponding **(slot_id)**$_c$ $\boxed{2}$. This number can be used as input for all other commands $\boxed{8,9,10}$, which either display data or steer the process. Therefore the slot becomes identical to the whole calculation and its **(slot_id)**$_c$ is at the same time also the 'calculation-ID', since the slot pools all relevant informations such as the directory and the runtime data, which are generated by the daemons and provide informations about how the calculation performs. The time-consuming tasks are performed by ⌜*JOBS*⌟ in the background $\boxed{4}$ and may or may not use the job-registry $\boxed{5}$ as a temporary storage container. In contrast to the previous example in Fig.2.6.4 the respective **(job_id)**$_C$ is not relevant to the user any more, since all data of interest have to be written to the slot-registry now 6, from where it can be retrieved as before 10 by the respective command, which requires the **(slot_id)**$_c$ as a mandatory argument. In this case now, the command also returns data if the calculation has already come to an end and the $\ulcorner JOB \urcorner$ has been deleted 7 from the table, because the calculation-data are persistent in the slot-registry. If desired, the calculation can also be rerun 9. For simplicity the presented example focuses on the GridClient-side of the workflow. Of course a remote calculation also needs a corresponding GridServer-slot, otherwise there would not be a temporary directory on the GridServer, but the respective **(slot_id)**$_S$ is irrelevant for the user and is one of many runtime-data, which are stored in the GridClient-slot. The pervasive purpose of this container becomes apparent if one takes into account, that the single client-side **(slot_id)**$_C$ also represents the server-side processes, its data and $\ulcorner JOBS \urcorner$, simply the whole calculation. The GridClient- and the GridServer-slots are slightly different and are discussed in detail in the following subsections.

2.6.5.1 GridServer

In order to start a calculation on a remote host, the GridServer needs access to the input files. These files may either be accessible directly without copying if the source code directory is available to the GridServer (NFS) or have to be copied to a local temporary directory. Additionally most GridServer-processes produce numerous logfiles, which are provided for debugging purposes but are mostly irrelevant otherwise. **W2GRID** imposes a policy, which demands, that these logfiles are written to a temporary directory on the GridServer first and only copied on demand. Therefore a temporary directory on the GridServer is needed anyway. The advantage of this solution is, that all temporary data, which is accumulated during the calculation is cleared when the slot is released. Table 2.3 shows the definition of the slot-registry. When a slot is reserved (**{slot.reserve}**$_S$), an additional entry is inserted into the table, which

column	*datatype*
slot_id	int
user_id	int
status	int
expiration	datetime
tempdir	char 150
workdir	char 150
parameter	text

Table 2.3: GridServer slots (table:slots.server)

is identified by the **(slot_id)**$_S$. In a second stage, the temporary directory **(tempdir)** $\boxed{3}_{2.5}$ is created in the directory ♯$GRIDROOT/temp/. Its (random) name is composed of the string 'slot_' and an attached integer. The **(workdir)** is by default the same directory as **(tempdir)**, unless it is specified explicitly upon reservation. Whereas **(tempdir)** is used for temporary data such as logfiles, the **(workdir)** points to the directory, where the files (input and output) are located. If the original source code directory is accessible (e.g. by NFS) and **(workdir)** is set accordingly no files have to be copied. During the calculation, the database-entry may hold a lot of additional data in the column **(parameter)**. The data and its formats are entirely up to the developer. Since a slot represents a calculation, the column **(status)** reports the state of this calculation. It may be one out of four integer values associated with a corresponding string: The default-status is 'INACTIVE' (0). If the calculation is started, it will be updated to 'RUNNING' (1). Finally it can either be 'FINISHED' (2) or 'ERROR' (-1). The sum of INACTIVE and RUNNING slots is limited and has to be specified during the installation. The default value is 6, but it may be changed freely to any other number. If this threshold is reached, no additional slot can be registered, hence the command **{slot.reserve}**$_S$ will return an error, unless one of the 'INACTIVE' slots is removed by **{slot.release}**$_S$ or a 'RUNNING' one comes to the final status 'FINISHED' or 'ERROR'. For failsafe and to avoid a shortage on slots due to application bugs, each slot has got an expiration date **(expiration)**, which is monitored by a job ($\ulcorner slot.check \urcorner_{S,C}$). If the slot is not updated until this date it will be changed to 'ERROR' and removed after a certain time. The content of the slot-registry may be retrieved by the use of **{slot.list}**$_S$ (list of all slots) or **{slot.info}**$_S$ (single slot). A sample output is given in Fig:2.21. The GridServer-slots are usually managed by the GridClient (see Fig.2.1). As explained in

```
>slot.list
#command took 1 seconds to complete
ID: [1] | reserved: [12:57:26 12-10-2006] INACTIVE
```

Figure 2.21: Screenshot of the command **{slot.list}**$_S$

chapter 4, at the beginning of a workflow the GridClient will select a host and reserve a slot. Subsequently it copies the input files and starts the calculation. Once the calculation is done, the GridServer-slot will not be needed any more and is released by the GridClient. If some of the logfiles are considered to be important, they have to be copied, otherwise they are deleted.

2.6.5.2 GridClient

In contrast to the GridServer, the slots are managed by the user directly by the aid of the commands **{slot.create}**$_C$, **{slot.kill}**$_C$ and **{slot.list}**$_C$. A temporary directory is not necessary in this case, since the working-directory is supposed to be accessible anyway and the logfiles are written directly into it. Hence there is only a single column provided **(dir)** which points to the source code directory of the calculation. The command **{slot.kill}**$_C$ does of course NOT remove this directory and just deletes the entry of the slot-registry instead. The definition of this table is given in Table 2.4. Additional informations can be provided, such as the **(program)** or a **(name)**, which allows the user to set the purpose of a calculation. The **WIEN2k** plugin uses this column for the name of the CASE (see 1.1.3.4). The entry for the **(server)** indicates, which GridServer received the calculation. For arbitrary and very calculation-specific data (e.g. the number of already completed iterations of **WIEN2k**, see chapter 4) an additional column **(parameter)** is provided. It may contain any kind of information, relevant to the user. Finally, the slots may also serve as a 'history' of all calculations performed so far. The screenshot in Fig.2.22 illustrates this purpose. Application-specific commands may also use more sophisticated methods to display these data since they can read and interpret the column **(parameter)** (see the output of **{wien.list}**$_C$ Fig.4.8).

column	*datatype*
slot_id	int
program	char 100
name	char 100
server	char 100
status	int
dir	char 150
parameter	text

Table 2.4: GridClient slots (table:slots.client)

```
>slot.list
#command took 0 seconds to complete
ID:[20] test            [FINISHED: WIEN2k on 'gescher'] /data/test
ID:[21] old tic         [ERROR   : WIEN2k on 'athena' ] /data/tic_old
ID:[22] tic             [FINISHED: WIEN2k on 'aurora' ] /data/tic
ID:[23] tio2            [RUNNING : WIEN2k on 'gescher'] /data/tio2
ID:[24] old tio2        [FINISHED: WIEN2k on 'gescher'] /data/tio2_old
ID:[25] random numbers  [QUEUED  : TEST   on 'athena' ] /data/random
```

Figure 2.22: Screenshot of the command **{slot.list}**$_C$

2.6.6 Registry

Previous sections already mention certain tables, which are termed 'registries' (e.g. 'job-registry', 'host-registry'), where the prefix (e.g. 'job-' indicates their content and purpose). The actual **registry** (without prefix!) of this section is a special table, which stores runtime informations about the daemons such as the port, the encryption-key and some configuration data. The definition of the **registry**, which is the same for both Perl-daemons is shown in table 2.5. Certain entries are mandatory and will be checked by the daemon at start-up 1 2.14

column	*datatype*
key	char 100
type	char 50 (contains either ’STRING’ or ’NUMBER’)
value	tinytext

Table 2.5: Table-defi nition of the registry

(e.g. ’host_port’, ’host_key’, ’host_ip’). The number of optional entries is not limited, hence this table can be used to store any kind of data (e.g. the configuration of the plugins). This becomes evident by comparing the sample contents of the GridClient-**registry** (Fig.2.23) and the GridServer-**registry** (Fig.2.24). The registry-tables are also needed by their respective commandline interfaces to retrieve IP, port and key. Details about the tables are provided in the following subsections.

2.6.6.1 GridClient

A sample content of the GridClient registry is shown in Figure 2.23. Since this daemon does not need to interact directly with the operating system or submit any calculations, the entries are limited to the items required to bind to a port. No configuration is required.

```
>select * from hostinfo.client
*-------------------------------------*
| key       | type   | value          |
*-------------------------------------*
| host_port | NUMBER | 9675           |
| host_ip   | STRING | 192.168.000.019|
| host_key  | STRING | POzisHb_IQdZ   |
*-------------------------------------*
```

Figure 2.23: A sample content of the GridClient-**registry**

2.6.6.2 GridServer

The configuration of the GridServer requires more effort, because this daemon must interact closely with the architecture and take care of the individual setup of given software components. To account for the long strings, a single configuration entry may create, the column **(value)** may take up to 8192 bytes. Most of these items are stored in XML-format for convenient machine-readability as shown in table 2.24, whose entries exceeded the line-width and had to be truncated.

```
>select key,type from hostinfo.server
*-----------------------------------------------------------------------------------*
| key                   | type   | value                                              |
*-----------------------------------------------------------------------------------*
| host_port             | NUMBER | 8833                                               |
| host_ip               | STRING | 228.130.134.45                                     |
| host_key              | STRING | SomeKey                                            |
| host_name             | STRING | athena                                             |
| slots                 | NUMBER | 6                                                  |
| free_slots            | NUMBER | 3                                                  |
| host_type             | STRING | standalone                                         |
| enable_authentication | NUMBER | 0                                                  |
| platform_id           | STRING | _PLAT_SUSE                                         |
| platform_options      | STRING | <options><item><name>platform</name><value>...     |
| processor_id          | STRING | _CPU_P4_2.4                                        |
| processor_options     | STRING | <options><item><name>SPEED</name><value>500...     |
| execution_id          | STRING | _EXEC_CMDLINE                                      |
| execution_options     | STRING | <options><item><name>cpus</name><value>1...        |
| execution_file        | STRING | /WGRID/libs_perl/execs/commandline.pl              |
*-----------------------------------------------------------------------------------*
```

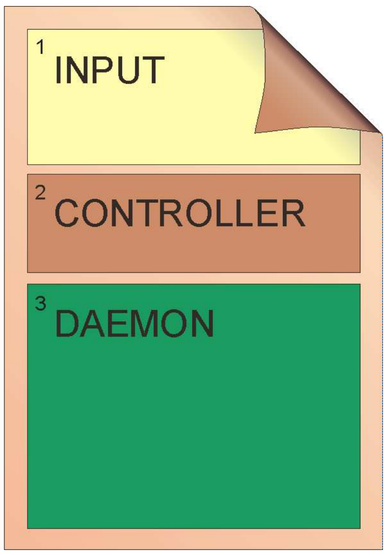
Figure 2.24: A sample content of the GridServer-**registry** (truncated)

2.6.7 Minimising the memory requirement of the Perl-daemons

If many users want to access the same host, there are two possibilities. Since the GridServer (as well as the GridClient) provides user-accounts[24], this feature can be used to share a single GridServer among several users. Or on the other hand, every user may install his own GridServer. **W2GRID** does not impose preferences for any of the two solutions. Given, that many independent GridServers are running on the same host, the most unfavoured disadvantage of Perl becomes apparent, because in comparison to compiler languages such as C or Fortran it consumes much more memory for the very same task [100, 101]. This property is even more striking, since the child processes always inherit the size of their parent, hence each child and grandchild will consequently be larger than its ancestor. The workflow of the Perl-daemons (Fig.2.14) makes apparent, that the processes fork quite often, therefore already a small reduction of the memory requirement will result in a reasonable saving of resources. As can be clearly seen in Figures 2.14 and 2.17, two processes are required for an idle daemon[25], one additional process for any connection[26] and two additional processes for every command being executed. This makes at least 5 processes at the same time for a single command such as **{wien.exec}**$_S$. It is necessary to reduce the memory consumptions in such a way, that the overall memory occupied by the Perl-processes remains below a certain threshold.

[24]which are disabled by default
[25]yet not serving any connection or executing any job
[26]e.g. different users

2.6.7.1 Breaking the daemons into smaller pieces



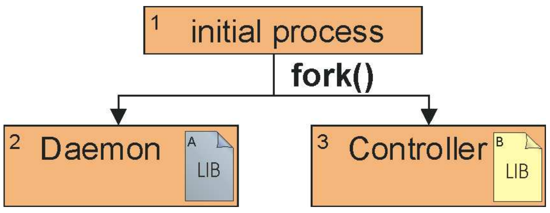
Each section is 'dead weight' for any of the other sections.

1. *Commandline arguments are processed. This code is only needed at start-up.*
2. *The code for the 'controller' of background processes (⌈JOBS⌉), which is illustrated in 4 2.14.*
3. *The actual daemon process, which waits for incoming connections. It is illustrated in 2 2.14.*

The additional libraries, which are needed for each single section in addition to the source of the daemon are not explicitly shown in this figure.

Figure 2.25: The three sections of the source code of Perl-daemons

Scripting languages are in general known to be excessively memory demanding, which does usually not matter for tools, that take a defined time to run. Yet in the case of daemons, which run for weeks and months, this is different. Their memory consumption must be as minimal as possible, which directs the attention to such lines of code, which are present in the sources but not used by the daemon although adding to its memory requirement. This becomes evident in Fig.2.25, which illustrates the composition of the respective source code file of a daemon. The section, which processes the input is used only once at startup 1 2.25 and for the many hours, days or even months while the daemon is online none of the therein provided functions will be executed again. 2 and 3 of Fig.2.25 are even less favourable, since they never use a line of code of each others section. The solution for this problem is simple: The code



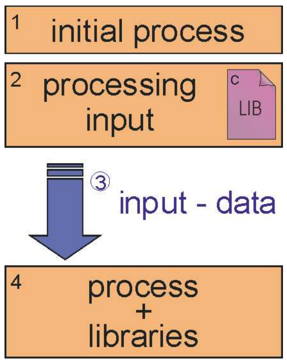
1 *The still small process at 'start-up'.*

2 *Library A is included into the daemon, which listens to the socket.*

3 *Library B is needed by the Controller, which processes the jobs.*

Figure 2.26: Optimisation of the child processes of a Perl daemon with respect to the memory consumption

is split into three pieces, according to Fig.2.25. While the first section 1 2.25 remains as the executable (***gridsrvd.pl***), the other sections are placed into the library files A 2.26 and B 2.26. These files are now included right after forking, which prevents, that the code of one instance contributes to the size of the other one. Still the input 1 2.25 is part of both processes, which is unfavourable. Therefore a third library C 2.27 is necessary, which consists of the code required for processing the arguments of the daemons. The 'dead weight' problem of this library is



1 *The original process* 1 2.26 *still without any additional code.*

2 *Arguments are read by the use of library* C*, which will here be included directly into the (one and only) parent process.*

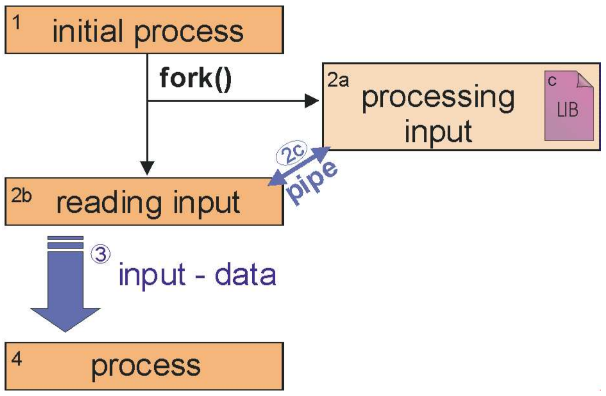
C *contains the code, which reads and processes the commandline-arguments of the daemon.*

3 *Only the extracted data shall be passed on.*

4 *All further processes do not need the functions of* C*, which is 'dead weight'.*

Figure 2.27: Unfavourable processing of commandline arguments

illustrated in Fig.2.27, where the file will still be passed on to all children of this process. The solution - shown in Fig.2.28 - is not trivial but allows to obtain the desired data without having to keep the respective library in the persisting code of the daemon. The presented method allows to 'load' and 'unload' additional code into and out of a Perl program by the use of two processes and interprocess communication. Only the input data is passed on to the parent process, but the library is used once and then 'discarded'.



The original process forks and uses the child 2a *to include the argument-processing library* C 2.27*. Once this is done, the already verified arguments* 2c *are sent to the parent* 2b *by the use of a pipe. The workflow will continue without any extra memory requirement. The child quits and its allocated memory is freed.*

Figure 2.28: Memory-saving processing of commandline arguments

2.6.7.2 Other dispensable libraries

The solution described in Fig.2.28 applies to the daemon process 2 2.14 and the control-process 4 2.14 and reduce the memory requirement of the Perl-daemon to an acceptable extent. Several other child processes such as the Bouncer 3 2.17 (see Fig.2.17) will also needlessly waste memory, although not performing any highly complex task. The procedure applied to these processes is similar and is not discussed here in detail.

2.6.7.3 Including modules

Perl comes with the favourable option to include foreign code[27] as a 'module' [102]. This is very convenient, because some tasks will better be solved with C-code, which requires less memory and is usually faster. A lot of functionalities such as the AES encryption are already provided in C-libraries, which can be directly included into the scripts, and helps to save many lines of Perl code, thus reduces the overall memory requirement.

2.6.8 Performance tuning

A major issue was the performance of the Perl daemons, which is especially critical if contacted from the commandline in batch mode[28]. If many subsequent commands are sent to the daemon (e.g. from a web interface, which cannot maintain an interactive session with the daemon), the logon process needs to be repeated for every single command individually and quickly becomes a significant overhead. In contrast to the improvement of the protocol, shown in section 2.3.4, this section instead is focused on improving the speed of the corresponding Perl-code. The - already optimised - protocol remains unchanged.

2.6.8.1 Replacing encryption by SSH-tunnelling

The encryption of the logon-process is optional but enabled by default. It can be disabled and replaced by SSH-tunnelling, which yields the additional benefit, that it encrypts the whole traffic. The corresponding tools are provided by the operating system and can employ more effective means, since they are optimised for the given architecture. Hence this solution is usually faster for short strings than the AES encryption.

2.6.8.2 Including C-code

Not only memory intensive but also time-consuming functionalities are best handled with C-code. There are two ways to use C-code in Perl-programs. The latter has already been mentioned in section 2.6.7.3.

[27] C, C++, perl

[28] batch mode means, that the interface is invoked only for submitting a single command, obtaining the result and quitting right afterwards

- Compiling the code as an application (C-tool) and invoking this application with the proper arguments by a 'system' statement[29] from within the Perl-code. This has the advantage, that the C-code does not contribute to the memory requirement of the Perl-code requirement. Yet it is slower since the internal shell causes a significant delay (several milliseconds per statement).
- Including C-code as a module and calling its functions from within Perl will provide a good performance, although this increases again the memory requirement. Modules have already been mentioned in the previous section.

Whereas the module solution will yield the better performance, the external code will be smaller in size. Both methods are offered for choice during the installation. By default, the C-code will be included as a module, but this setting can be modified for the sake of memory consumption at any time. The usersguide [97] explains how to compile the module properly and restart the Perl-daemons to make the new environment variable take effect.

2.6.9 Logon

In contrast to the full protocol, which is employed by the wiensql-daemon and is explained in section 2.3.4, the Perl-daemons do not query for login and password by default and instead use the encryption key to authenticate a client. The user-authentication can be enabled with the aid of installation scripts, which also provide means to create user-accounts (i.e insert and manage entries to table 2.6). Enabling user-authentication will slow down the logon-process,

column	*datatype*
user_id	int
login	char 100
password	encrypted 100
full_name	char 100
description	tinytext

Table 2.6: Essential columns of the user-registry

which may be a problem if many commands are submitted in batch mode. The screenshot provided in figure 2.29 illustrates the default logon process without login or password being asked by the server. This verbose output is collected from the ***gridsrv_console.pl***, which connects to the GridServer, hence the screenshot shows the client's perspective on the logon-process. The initial **Greeting** is shown in line 6. The 'u' before the version-number indicates, that the

[29]A Perl-command, which allows to create an intrinsic Bourne-Shell (sh) for the execution of shell-commands.

```
1:  [SOCKET]   Trying to connect to localhost on port 8811!
2:  [SOCKET]   Socket creation succeeded
3:  [SOCKET]   Bind succeeded
4:  [SOCKET]   Connect succeeded
5:  [INTERNAL] sending greetings
6:  [SOCKET]   reading 'GRIDSRVD u.2.12 encrypted'
7:  [INTERNAL] connection to gridserver version '2.12'
8:  [INTERNAL] Gridserver will authenticate you as admin
9:  [INTERNAL] Gridserver is encrypted ...
10: [SOCKET]   reading '************************'
11: [INTERNAL] READING ENCRYPTED: 'tellme:9946%1430+2340-9011+2783*5467' [4]
12: [VERBOSE]  Latest Transmission was considered type:4 (everything >=0 is good)
13: [INTERNAL] SENDING ENCRYPTED: '15209356'
14: [SOCKET]   sending '******'
15: [SOCKET]   reading '************'
16: [INTERNAL] READING ENCRYPTED: 'access granted'
```

Figure 2.29: The verbose output of the commandline interface ***gridsrv_console.pl*** showing the communication between client and server during the default logon process

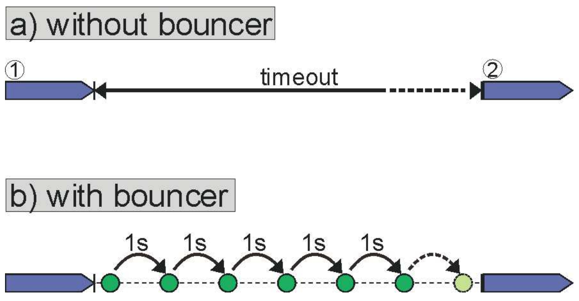
server will not ask for login and password, otherwise it would send an 'a'. Right after processing the greeting, this is noticed by the client on line 8. The encrypted **Maths-test** is received on line 10[30] and the result returned on line 14. The encrypted **Result** may either be 'access granted' (line 16) or 'access denied' followed by the respective reason.

2.6.10 The Bouncer

`W2GRID` does not impose any strict rules concerning the development of the command-scripts, hence some of them may take extremely long to be processed or even crash and cause unpredictable results. To provide a fault tolerant framework for commands and to catch these problems -especially the timeout of the pending connection due to an exceeding time-consumption - an additional child process $\boxed{3}_{2.17}$ is introduced. It will send short 'pings' in intervals of one second. As long as these pings are transmitted, the client which waits for data at the other end of the connection considers the remote process alive and does not close the socket, because the TIMER will be reset each time a packet arrives. This child process is referred to as the 'Bouncer'. The 'keep alive strategy' is illustrated in Fig.2.30. It is however only reasonable to keep a connection open as long as a command is being processed. If the command-instance $\boxed{5}_{2.17}$ is done with the workflow, the Bouncer will terminate and return the control of the socket back to the parent. Also, if this instance dies before it can actually return any response, which might be caused by a severe error in the script, the Bouncer will terminate, too. This triggers the exception handling of the client, which closes the connection safely. A regular check $\boxed{1}_{2.31}$ if the parent still exists makes sure, that the failsafe-mechanism works even in the case of a fatal error. Program-flaws like an endless loop however cannot be caught by the Bouncer, since the parent will still exist. There is no mechanism available to distinguish such a bug from

[30]line 11 shows the same line in plain text

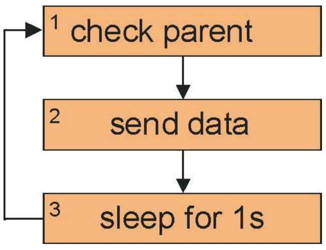
a really time-consuming process.



- ***blue arrow*** *The command-process* 5 2.17 *writes to the socket.*
- ***black line*** 5 2.17 *is silent.*
- 1 *starting the command workflow.*
- 2 *command workflow is done.*
- ***green circles*** *Bouncer packets.*

Figure 2.30: Socket-traffi c between client and server ('keep-alive strategy')

The child, which processes the command workflow will not write data to the socket unless the processing is done 2 . In Figure:2.30 a) the situation is illustrated without a Bouncer. After a threshold of several seconds, during which the child process is working through its instructions and remains silent, the client will consider the silence to be caused by a fatal error and close the connection. To prevent this unwanted interruption, the Bouncer sends small packets in regular intervals (see Fig:2.30 b). The content of these packets is not important as long as this part of the traffic can be distinguished easily from the output of the command-instance. If the command-instance dies, due to a fatal error in the script, its parent 6 2.17 will



1 *If the parent process does not exist any more, the Bouncer will quit instantly.*

2 *Even a single character is sufficient to keep the connection open. These packets can be used as carrier for other purposes (see later).*

3 *The interval in between two Bouncer cycles is approximately one second. The threshold for timeout is in the range of 5-10 seconds.*

Figure 2.31: Bouncer-workfbw

notice it and kill the Bouncer. For failsafe, the Bouncer itself also checks with each cycle, if its parent still exists. This double-failsafe procedure catches even the unlikely case that a crashing command-workflow kills the parent and would keep the Bouncer -now an orphan-bouncing forever.

2.6.11 The Progress Indicator

Whereas the minimum latency between submitting a command from the commandline to a daemon and receiving the result is in the range of a fraction of a second, the maximum latency is practically unlimited, since the workflow is up to the developer and the connection will not be closed thanks to the Bouncer. A user will grow anxious already after a few seconds if no output is displayed. resulting from the way, the RPC-commands are implemented, it has to to be completed up to the very last line of the workflow until the first byte of the result can be sent. Sending partial or even intermediate results is not supported. Hence, **`W2GRID`** needs a mechanism to inform the user about the progress of the command. For this purpose the **progress indicator** is amended to the data-transmission protocol. The progress may be displayed at the commandline in %.

2.6.11.1 Implementation

The **progress indicator** is a method to send such progress feedback packets to the client. Since the **Bouncer** (see section2.6.10) already constantly sends data to the client, and because the content of these packets is not important, they are hijacked to serve for an additional purpose. Hence, with the aid of the Bouncer, a constant feed of progress-packages are offered in intervals of a single second. A code fragment, which shows the implementation of such a progress report in a command script is shown in Fig.2.32 A list of ten filenames (a-j)

```
1: my @files=("a","b","c","d","e","f","g","h","i","j");
2: foreach(@files)
3: {
4:     &copy_file($_);
5:     &demon__progress(10); #10%
6: }
```

Figure 2.32: A sample code for including progress indication into the fi letransfer

is created as an array in line 1, which are copied individually in line 4[31]. Assuming all files have got the same size, the overall workflow will progress by 10 percent with each file, hence the respective progress (also in percent) is sent in line 5. The result, which is extracted from the Bouncer-packets in the meantime by the client is shown in Fig.2.33 The packets are (in-

```
[SOCKET] reading '<*>10</*>'
[SOCKET] reading '<*>20</*>'
[SOCKET] reading '<*>20</*>'
[SOCKET] reading '<*>20</*>'
[SOCKET] reading '<*>20</*>'
[SOCKET] reading '<*>10</*>'
```

Figure 2.33: Verbose output of the commandline interface ***gridsrv_console.pl*** showing the packets as received from the Bouncer process

valid) XML-fragments, which can be distinctly identified and ignored in the final string. The figure will yield a total sum of 100%, however there are only seven lines instead of ten, since some lines contribute a progress of 20%. This is due to the implementation, which implies that

[31]This is a dummy function, which does not exist. It only serves for illustration.

the progress is passed on from one instance to the next one (parallel-process -> Bouncer -> GridClient daemon -> commandline). This is only possible by the aid of files. The Bouncer collects these files with each cycle, sums them up and forwards the progress. If in the meantime according to the above example, two of these packets have been written to the respective temporary directory, the Bouncer will add their contributions and send only the result instead of each individual progress-packet.

2.7 C-Shell scripts

For the installation and configuration of **`W2GRID`**, several standard Unix/Linux commands need to be nested within syntactical elements. This yields simple straight forward workflows, which can be easily developed and modified. **`W2GRID`** comes with two master-scripts to perform the installation, which are described in more detail in chapter 3. One will invoke a fully script-based installation, whereas the other will offer a menu-guided interactive installation. All other shell-scripts are invoked from within the master-scripts and do not have to be called manually. The purpose of the provided routines is as follows:

- **Installation:** Creation of Environment variables, Modification of the shell startup-files[32], compilation of the source code[33], Creation of temporary directories, menu-guided setup of the daemons.
- **Modification:** Installation of new plugins, source updates, change of runtime-parameters[34], registration of new hosts.
- **Removal:** stop of the daemon processes, removal of Environment variables, restoration of the original shell startup-files, removal of temporary directories.

2.8 C and Perl-tools

Some tasks are not or only barely accessible within the scope of the C-Shell language like AES encryption or tcp functions. Also complex workflows, which require more than a single database-operation or occur several times in different C-Shell scripts are better solved with Perl, since wiensql can only be accessed from within Perl-programs. The plugin-installation routines will be encapsulated by C-Shell code, but the actual workflow is written in perl. Encryption and decryption on the other hand is supplied by C-programs. Only a few important

[32]This fi le is executed when a new shell is created. The C-Shell for example needs the .cshrc fi le, whereas the bash will execute the /.bashrc oder ./profi le fi le

[33]database and C tools

[34]e.g. the port or the encryption-key(see registry and security)

examples are itemized, the others are described in the usersguide [97] and the developers-guide [103].

- ***rijndenc*** AES encryption of files (rijndael). The rijndael algorithm has been implemented as a C-library. The binary makes this library available to parts of **W2GRID**, which cannot include the C-code neither as library nor as module.
- ***rijndec*** AES decryption
- ***bintest*** allows to perform a quick check, if the compiled binaries work. This is used by the installation scripts in order to check if the compilation has been successful
- ***query_password*** hidden password input
- ***filehash*** Generates a 16-100 bytes (base64 encoded) hash-key for a file. Developers may use this key to check if a file has changed by simply comparing the keys from before the change and after the change. It is a bit-by-bit algorithm, which must work on every machine and yields the same key for the same file for any kind of operating system (little-endian and big-endian).
- ***logview.pl*** Turns the XML-format of logfiles (see section 6) into a readable output.
- ***logalizer.pl*** Reads the logfile of the wiensql-daemon and returns a statistics of the exit-codes.
- ***gridpackage_create.pl*** Packs source code files, installation and removal- instructions into a single file, which can be used to distribute e.g. plugins or bug fixes (see section 3.3.11).
- ***gridpackage_install.pl*** Used for installing such a package (see section 3.3.11).

2.9 Core and plugins

Portable code has two different aspects. One is the portability to a given architecture, which can either be achieved by the use of a platform-independent scripting language (e.g. python, Perl) or by providing the source code binaries[35].
The other aspect is the capability to cope with the given configuration of a host and to interact with other applications or tools. This is especially important for such essential workflow elements like running of a calculation and its frequent monitoring. Starting a calculation requires to know the job-submission method, which is applied by the host in question. Since there are numerous methods, it is out of question, that the respective commands can ever be hard coded directly in the source code.

Example:
Given two clusters: One is managed by PBS and another one by the LoadLeveller. To start a calculation on PBS, one needs to write a submission script, which is different to the LoadLeveller script and submit it by the use of the command

>***qsub pbs.script***

in contrast to

>***llsubmit ll.script***

on the other cluster.

Even if the same cluster management software is used, the submission-script may look different due to different versions of the software or due to certain individual properties of the host such as the number of CPUs per node or the number of cores per CPU or simply due to the queue-names. Additionally the different methods of how to submit a calculation need to correspond with the given operating system. For this reason all such tasks, which require the use of very specific commands and an individual formatting due to the configuration of a host are abstracted in **W2GRID** as so called 'plugins'. This feature is illustrated for a task like the job-submission in Fig.2.34.

Example:
{wien.exec}$_S$ *starts a* ***WIEN2k****-calculation on the host. If this host is a PBS-managed cluster, it has to be submitted by the aid of the* ***qsub*** *command as shown in the previous example. The corresponding file* wien/exec.pl† *will not contain this command. Instead it will invoke a*

[35]**W2GRID** actually employs both

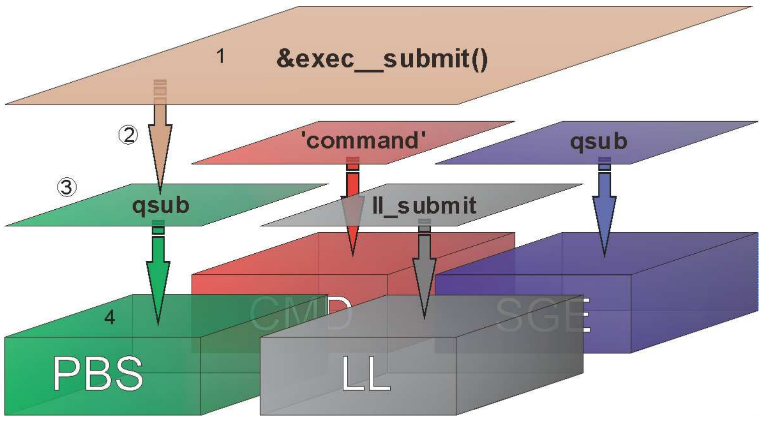


Figure 2.34: Abstraction of the job-submission by the use of plugins

so-called 'plugin-function'

The proper **plugin-function** for submitting a command is &exec__submit(), which is pre-defined and has a uniform formatting of the input as well as the output. The command-script uses this function 1 2.34 instead of explicit commands like ***qsub*** or ***llsubmit***. Each plugin-library contains exactly one such function. The actual implementations 3 are different among the individual files and conform to the specific needs of the submission-scheme. In order to know, which library shall be used, this has to be specified upon installation of the GridServer. The individual properties such as the memory of each node have to be configured. Each time, a command calls the function &exec__submit(), it will be mapped 2 to the appropriate method (e.g. 'qsub').
The library files are referred to as the 'plugin-library' or simply the **plugin**. The same principle is employed for all aspects, which need an individual architecture-dependent implementation. This **core and plugin**-approach allows to use of **`W2GRID`** on virtually any Unix-like architecture.

The **core** is by far the largest part of the **`W2GRID`** code and is stripped of all parts which require an explicit implementation of the items listed above. It provides the framework and the interfaces for including the plugins in a transparent way. The core is independent from any of the plugin- capabilities and is reduced to the parts, which can be compiled and run on any host. Only the Perl-daemons and the Perl-tools require this special treatment, consequently all C- and C-Shell code is part of the **core**. All plugins are restricted to providing explicite methods only for their specific purpose. Execution plugins only code the methods for interacting with the queuing system but do not interact directly with the platform. If for example the total memory has to be retrieved, the execution plugin has to use the corresponding mapped platform plugin function instead of directly calling one of the many possible commands for the different operating systems. Therefore the plugins remain independent of each other, which allows that each plugin is coded exactly once.

Example:
The Sun-Grid-Engine SGE may be used on different operating systems. The same SGE plugin which is used for Linux, can also be used on a Solaris cluster, since all platform-specific flavour is contributed from the platform plugin, and the execution plugin does not have to care.

The types of plugins are discussed in further detail in the following sections

2.9.1 Operating system (platform) plugin

Some ’standard’ Unix/Linux functionalities do not always use the same arguments or output such as ***ps*** or ***top***. Others do not even have the same command-name or be achievable by the use of a single tool (e.g. retrieval of the free memory or formatting of local dates). The operating system plugin is also referred to as the ’platform plugin’

already provided:

- SuSE
- Solaris
- Redhat
- AIX

Two code fragments are provided, which shall illustrate the functionality. The implementation of a simple functionality like the retrieval of the total memory of a host is shown for two different platforms. In both case a string, obtained from invoking different commands is processed by different regular expressions to extract the property of interest. The code is truncated to its elemental part.

2.9.2 Job-submission (execution) plugin

This is also referred to as the ’execution plugin’, because it does not only address the job-submission, but also all other tasks associated with running applications, like monitoring and status information retrieval.

```
sub totalmem
{
    my $aix=‘lsattr -E -lmem0‘;
    if($aix=~/goodsize ([0-9]+)/)
    {
        return $1;
    }
    return 0;
}
```

Figure 2.35: Code snippet for retrieving the total memory on AIX

```
sub totalmem
{
    my $suse=‘cat /proc/meminfo‘;
    if($suse=~/MemTotal:\s+([0-9]+)/)
    {
        return $1;
    }
    return 0;
}
```

Figure 2.36: Code snippet for retrieving the total memory on SuSE Linux

already provided:

- LoadLeveller (LL)
- Portable Batch System (PBS)
- Sun Grid Engine (SGE)
- Commandline

2.9.3 Connection plugin

This plugin defines, how commands are submitted to a server and the respective results are received. It serves basically for all data transfer except the filetransfer. The client has to contact the server and establish a connection first. This process does not always follow the same scheme. If the ports in question are blocked by a firewall, one can use SSH-tunneling, which forwards a local port to the SSH-port, but requires completely different methods for initialisation than the usual tcp socket connection. It is yet more difficult if other middleware is involved. The GLOBUS Security Infrastructure for example needs a proper certificate-handling and to tunnel commands trough third-party executables. All these different issues can only be addressed by using a proper plugin.

already provided:

- socket (tcp)
- SSH-tunneling

2.9.4 File transfer plugin

Different to the connection plugin, this one is needed exclusively (as the name suggests) to copy files from and to a server.
already provided:

- cp
- rcp
- scp

2.9.5 Application plugin

The application plugins add the capability to control a certain application by the use of **W2GRID**. This requires however, that the application itself has already been installed properly in advance. The development of application plugins in general is explained in section 3.3.5. The **WIEN2k** plugin is illustrated in detail in chapter 4.

2.9.6 Processor plugin

The client-side methods of an application plugin usually contain strategies, how to select the proper resources out of the pool of registered GridServers. This in turn requires, that the GridServers are able to predict the runtime of an application, which depends on the power of the CPU. To measure this power, **W2GRID** uses an arbitrary number, often simply referred to as the 'speed', which is a benchmark number in the range of 200-1000 at present. The user does not know this number, and it is not very convenient to supply a list, from which one could select the number and enter it manually. Instead, the number is supplied from this plugin. The processor plugin just comes with a single XML-file, which defines the name of the chip, a short description and the number. Additional scripts like those necessary for all the other plugin-types are not required.

2.9.7 Plugin development

A plugin is composed of **commands and jobs**, **tools (in any language)**, **library file(s)** and in most cases an XML-**configuration-file** (see a sample content in Fig. 2.37). The latter one is required to query informations from the user like the version-number of the operating system. Application plugins may be implemented freely and do not even need the plugin-descriptor,

```
<PLUGIN>
        <ID>_PLAT_AIX</ID>
        <TYPE>PLATFORM</TYPE>
        <NAME>Aix</NAME>
        <CUSTOM>
                <ITEM>
                        <NAME>platform</NAME>
                        <REGEXP empty=yes default=unknown>STRING</REGEXP>
                        <DESCRIPTION>The AIX version</DESCRIPTION>
                        <QUERY>Which version of AIX do you use:</QUERY>
                </ITEM>
        </CUSTOM>
        <FIXED>
                <ITEM>
                        <NAME>system_file</NAME>
                        <VALUE>libs_perl/platforms/aix.pl</VALUE>
                </ITEM>
                <ITEM>
                        <NAME>endian</NAME>
                        <VALUE>1</VALUE>
                </ITEM>
        </FIXED>
        <DESCRIPTION>IBM operating system</DESCRIPTION>
</PLUGIN>
```

Figure 2.37: A sample content of a plugin confi guration-fi le (extracted from pbs.plg)

whereas the others are more or less constrained by their very specific nature and impose, that the libraries offer certain functions, which are well-defined by their name, input- and output-format.
The development is explained in detail in chapter3.

2.9.8 Interoperability

Most newly developed middleware lacks a proper interoperability with existing ones. **`W2GRID`** addresses this issue by abstracting the functionalities of other middleware simply to a method of job-submission, filetransfer or connection, which can be picked up by the proper plugin. If a cluster for example is entirely built on GLOBUS tools, submitting jobs to this cluster can only be done by the aid of commands like ’globus-job-run’. This is not very different from the way, how jobs are submitted to any other queuing system like PBS. This possibility has been explored only in theory. A proof-of-concept is in preparation.

2.10 W2GRID Component packages

Distributing code for **W2GRID** (e.g. plugins, bug fixes) often requires more than simply providing an archive, which contains the source code files. Prior to the installation of an additional part it may be desirable to perform some checks, if a certain list of pre-requirements are met (e.g. if certain directories exist). Some tools may be supplied as source code instead of binaries, and have to be compiled. Bug-fixes have to replace components and maybe need to alter table-definitions. Providing the source code and an ASCII file (README.txt) for the user is not a reliable and convenient approach. Instead **W2GRID** provides an automated scheme, which executes such instructions that are supplied by a script file. A 'package' is a single binary file, which consist of the tared and compressed source code files and also contains a set of XML-coded instructions, which are supposed to be interpreted and executed by a certain tool (***gridpackage_install.pl***). Some of these instructions also rule, how this package may be removed again or how a certain bug fix can be undone. The structure of such a package is shown in Fig.2.38.

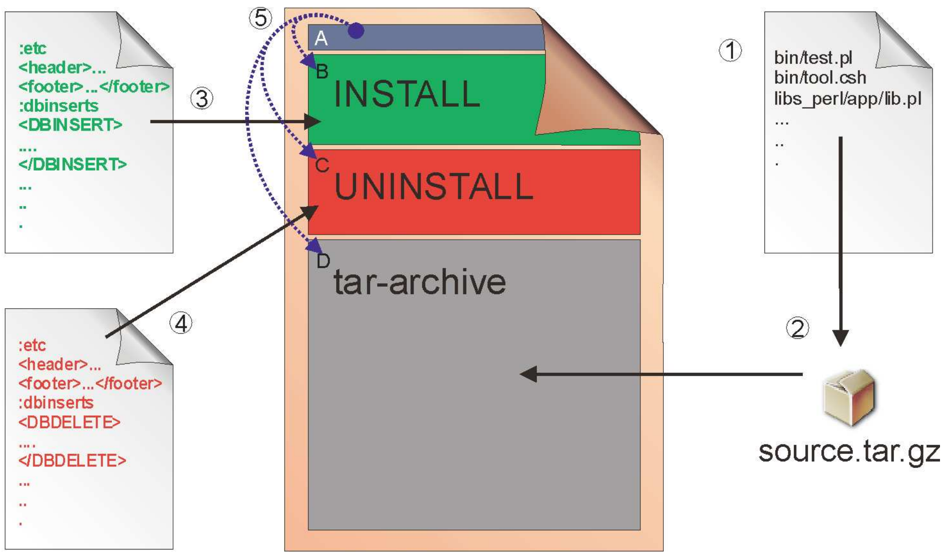


Figure 2.38: Contents of a **W2GRID**-package fi le

1 A list of the files, which belong to the package have to be supplied as a simple ASCII file with relative pathnames. (see a sample file in the appendix Fig.A.18). The respective files are compressed into the archive 2. Files are optional, since a bug fix may for example only contain a simple regular expression instruction in 3 to correct a piece of code, hence also this list is not mandatory.

2 The tared and compressed archive becomes attached to the end of the package D

3 The installer file is XML coded and separated into several sections. A sample file is shown in the appendix (Fig.A.16). It is appended as the second section to the package B. The file is optional too, but either this or 1 has to be supplied to the package creation-tool, otherwise it will return an error.

4 The package may also provide removal-instructions, which allow to revert the installation process. A sample file is shown in the appendix (Fig.A.17). It is appended as third section to the package C and is of course optional too.

5 The header section A of the package file contains the sizes of the respective parts B and C in order to separate the sections again upon installation.

The purpose of the package-feature of **W2GRID** and the creation of own packages for e.g. sharing application plugins is shown in section 3.3.11. The tasks, which are supported in the installation/removal scripts are as follows:

- Creation/removal of additional databases
- Creation/removal/Check of tables
- Creation/removal of directories (e.g. for command-groups or job-groups)
- Compilation of sources and copying of the resulting binary to ♯$GRIDROOT/bin/
- Inserts/Updates/Deletes to the wiensql-database
- Check for environment variables.
- Interactively query dynamic variables and use the results as conditions for the installation/removal workflow.
- Check for installed programs/versions
- Insert/Delete of a corresponding package-entry to the database
- Registration of plugins

Chapter 3

Using and extending the middleware

The predominant purpose of **`W2GRID`** is to turn numerous user-accounts on different resources into the private Grid-environment of a scientist, which has to be installed to a large extent by the user in person. Since the GridServers need to interoperate properly with the architec-

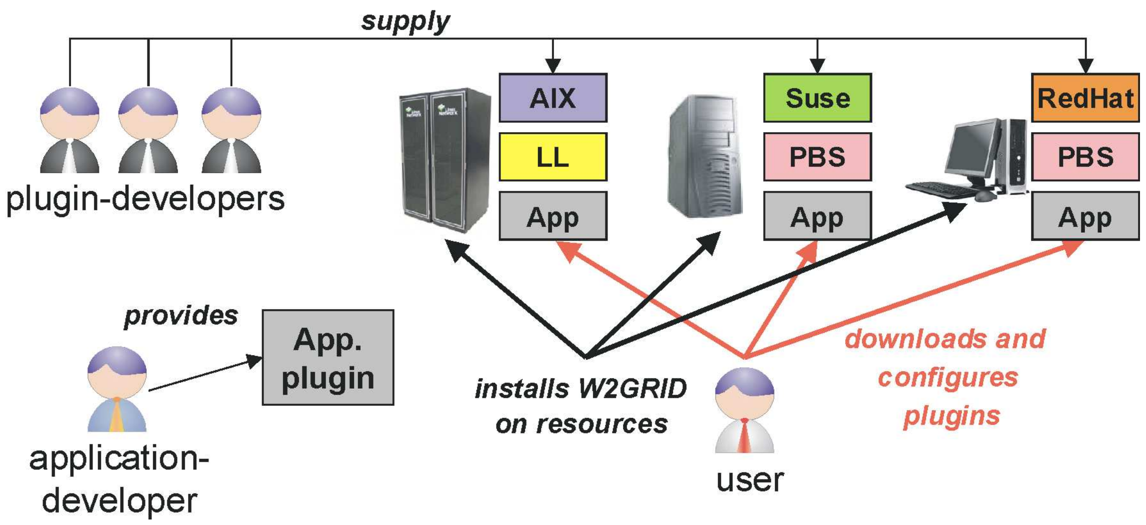


Figure 3.1: Collaboration of developers and users of **`W2GRID`**

ture and the setup of their host, a number of plugins has to be installed and configured. A basic set for the most common platforms and queuing systems are already provided, whereas additional plugins may be developed and published at any time by any person, who wants to contribute to **`W2GRID`**.

In order to use a certain software by means of the presented middleware, one needs to have an 'application plugin', which consists of scripts that 'wrap' the software and allow it to be run on the infrastructure. In most cases these scripts are provided by the developer(s) of the application. The users obtain it either with the distribution of the HPC software or download it from the Internet. The three collaborating parties and their responsibilities within the **W2GRID** framework are illustrated in Figure 3.1.

Remarks to this chapter
The detailed informations provided in this chapter are not intended to serve as a usersguide for installation and maintenance, as this exceeds the scope of what one needs to know about **W2GRID**. Furthermore the tasks to be performed for installing the middleware on computing resources are described in the first sections of [97]. Instead the chapter shall provide the developer with a profound understanding of the many processes, that are performed in the background invisible to the user.

3.1 Installation

W2GRID is installed by the aid of several csh-scripts, which may either be used in an interactive (or manual) mode (section 3.1.3) or by the aid of config-scripts (section 3.1.2). Both methods offer basically the same flexibility and options for the installation, yet the manual mode is the only way to modify the setup afterwards, whereas the installation based on config-scripts is intended to be applied for installing **W2GRID** completely from scratch. Most details of either mode are hidden from the user, hence this section will highlight the processes, that are performed in the background (section 3.1.1). Extended informations and troubleshooting-tips are provided in the usersguide [97], yet the essentials are summarised in this section for clarity and completeness of the thesis.

3.1.1 Task-overview

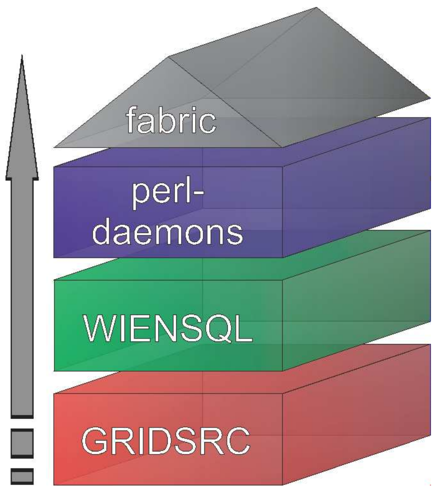
The extent of user-interference required for the individual tasks is different. Some need only little (or even no) interaction (e.g. compilation) whereas others, such as the configuration of the GridClient and the GridServer, usually comes with a list of details about the architecture, which have to be specified interactively by the user. The installation covers the following tasks:

- Creation of 4 environment variables and modification of the startup-file(s)[1].
- Creation of two directories ($GRIDROOT and $WIENSQL_ROOT).
- Compilation of the C-source code.
- If required, the interpreter-path of the Perl-scripts has to be changed.

[1]The C-Shell is used as internal default-shell, hence the .cshrc fi le has to be provided anyway. If the bash is used, the modifi cations need to be done to both startup-fi les.

- Installation of the database.
- Installation and configuration (plugins) of the GridServer daemon on each computing resource.
- Installation of the GridClient daemon (only on a single host) usually on the local desktop.
- In a last step, the GridServer daemons have to be added to the host-registry, which gives the infrastructure the intended shape (see Fig.2.1). The required data are **(host_ip)**, **(host_port)**, **(host_key)** and optionally also a pair of login and password if the user-authentication is enabled. After configuration of the connection plugin (section 2.9.3) and filetransfer plugin (section 2.9.4), the GridClient will retrieve all further static data (e.g. memory, installed applications) autonomously.

The individual sections of the infrastructure setup are referred to as 'layers' (see Fig.3.2), which depend on each other and have to be installed in the presented hierarchical order (arrow).



- ***GRIDSRC:*** *Compiles all executables and makes the Perl-binaries work. Directories and environment variables are created. The Infrastructure is not operable, but individual parts may already be used.*
- ***WIENSQL:*** *Installs the wiensql-database and starts its daemon. The infrastructure is still not operable, but the database may already be used.*
- ***Perl-daemons:*** *Either a GridServer or a GridClient daemon has to be installed to make this host part of the infrastructure.*
- ***Fabric:*** *After having installed either one or both daemons, the respective resource needs to be integrated into the infrastructure (see Fig.2.1). The corresponding shell-script provided for this tasks is only available in an interactive mode.*

Figure 3.2: The task-layers of a **`W2GRID`** installation

3.1.1.1 GRIDSRC

This layer will yield fully functional binaries and Perl-executables of **`W2GRID`**. It is useful for testing the success of the compilation and the functionality of the installation scripts on the host in question. The two Perl-daemons and the wiensql daemon are not usable yet, nor any script which either needs wiensql-access or has to connect to one of the daemons (e.g. commandline interfaces). To stop the installation after completion of this single layer (e.g. for debugging purposes), the quick-installation has to be invoked with the command:

>***./quick_install.csh GRIDSRC***

The input is read from the file gridsrc_autoinstall.conf† (see Fig.3.3). The following tasks are performed:

- Two environment variables -$GRIDSRC and $GRIDROOT- are added to the startup-file (see lines 12 - 13 of Fig.A.19 in the appendix).
- The directory ♯$GRIDROOT/ and its many subdirectories c 2.5 are created, which will contain all compiled binaries, libraries, the Perl-module(s) and temporary data. It is by default located at ♯~/.wgrid_$hostname/[2].
- Some essential paths and host-properties are retrieved autonomously and used in order to adjust certain settings in the numerous Makefiles (e.g. compiler name).
- The libraries and binaries are compiled and written to the directories ♯$GRIDROOT/libs/ and ♯$GRIDROOT/bin/.
- If the optional modules have been enabled, they are compiled and copied to a subdirectory of ♯$GRIDROOT/libs/.
- An additional environment variable ($PERLLIB) is either added or modified. Since the Perl-module is optional, the variable is optional, too.
- The directories ♯$GRIDROOT/bin/ and ♯$GRIDSRC/bin/ will be attached to the path-variable.
- The correct path to the Perl interpreter will be written to the shebang-line[3], if it differs from the default.

Since some changes have been made to the startup-file, it has to be reloaded to make the changes take effect.

3.1.1.2 WIENSQL

The wiensql-database, which is installed in the second layer is required by the two Perl-daemons and most of the tools. To stop the installation, right after completing the database-installation, ***quick-install.csh*** has to be called with the command:

>***./quick_install.csh WIENSQL***

[2]This is the hostname of the local computer without the domain-name attached.
[3]The fi rst line in the script. It usually looks like this '#!/usr/bin/perl'. See http://faq.perl.org/perlfaq7.html

A fully operable (and in most cases already running[4]) database-daemon is obtained. Its tools such as the wiensql commandline interface (***wiensql.pl***), which can be retrieved in the directory ♯SRC_wiensql/bin/ may be used now. The Perl-daemons on the other hand are still not operable, because none of the required tables exist yet. The input for ***quick_install.csh*** is read from the file wiensql_autoinstall.conf†. The following tasks are performed:

- An additional mandatory environment variable ($WIENSQL_ROOT) is created (see line 10 of Fig.A.19 in the appendix).
- The corresponding directory ♯$WIENSQL_ROOT/ is created, which by default is located at ♯~/.wiensql_$hostname[5]/.
- The runtime-parameters are written to the configuration-file .wiensqlrc†.

Again some changes are made to the startup-file, that require it to be reloaded in order to make the changes take effect.

3.1.1.3 Perl-daemons: GRIDSRV, GRIDCLIENT

The pre-requisite for installing this layer is a working database[6]. The type of the daemon ('GRIDSRV' or 'GRIDCLIENT') has to be specified at the commandline.

>***./quick_install.csh $daemon***

installs only the respective $daemon, whereas the command

>***./quick_install.csh GRIDSRV,GRIDCLIENT***

installs both. The following tasks require only little user-intervention:

- Creation of a wiensql user-account.
- Creation of the necessary tables and insertion of essential data (e.g. **(host_ip)**, **(host_port)** or **(host_key)**).
- Registration of available plugins. All XML plugin-descriptors are read and their data inserted into the plugin-registry.

The subsequent configuration of the daemons requires a certain user-knowledge and may only be performed interactively. For this purpose the script ***quick-install.csh*** will launch the manual installer, as soon as the installation reaches this point. The performed tasks focus primarily on the configuration of two GridServer-specific plugins, which are mandatory:

[4]depends on the settings in the confi g-fi les
[5]The hostname without the domain-name
[6]Both daemons will use separate accounts.

- Install and configure the proper platform plugin for interaction with the operating system.
- Install and configure the execution plugin. The informations about the type of the CPU, the size of the memory, the number of nodes/CPUs/cores are mandatory (only if these data cannot be retrieved automatically). Depending on the plugin additional data may be required (e.g. the command for submitting MPI-parallel jobs). A more detailed explanation of the available options of execution plugins are provided in the users-guide [97].

Since no further changes are made to the startup-files, they do not have to be reloaded after installation/update of any of the Perl-daemons. Additional application plugins may be installed and removed at any time.

3.1.1.4 Fabric

In a final step, the individual GridServers must be registered as resources to the GridClient in order to obtain a working **W2GRID** infrastructure. This feature is not available from within ***quick_install.csh*** and has therefore to be performed by the aid of the manual installer:

- A **(host_name)** for identification[7] is required.
- The **(host_ip)** and the **(host_port)** must be supplied, afterwards the script tries to contact the GridServer in order to verify the given input. Since this test employs the default connection plugin (socket), which does not work through firewalls, this first shot may fail and can be ignored.
- For the logon, the client also needs the **(host_key)**, which is probed in a second connection attempt. The same reasons as before may cause the attempt to fail, which can be ignored, too.
- Depending on the setup of the GridServer it might also be required to authenticate a user by login and password. The input is written to the host-registry of the GridClient and does not have to be provided again.
- Finally the GridClient must be told, how the remote daemon can be contacted. A list of connection plugins is displayed in a list. The user has to select the proper one and enter a few additional details such as the remote login-name.
- In the same manner, the filetransfer plugin is selected and configured.

Provided the GridClient daemon (and at least one GridServer daemon) is online, the personal Grid-infrastructure is ready for use.

[7] does not have to be the actual 'hostname'

3.1.2 Quick Installation

The input is read from configuration-files and does not have to be provided manually. The process is actually performed by four separate subscripts, which are called subsequently from within the master-script as illustrated in Fig.3.3. Whether the one or the other script is invoked depends on the input.
The installation is started upon the command:

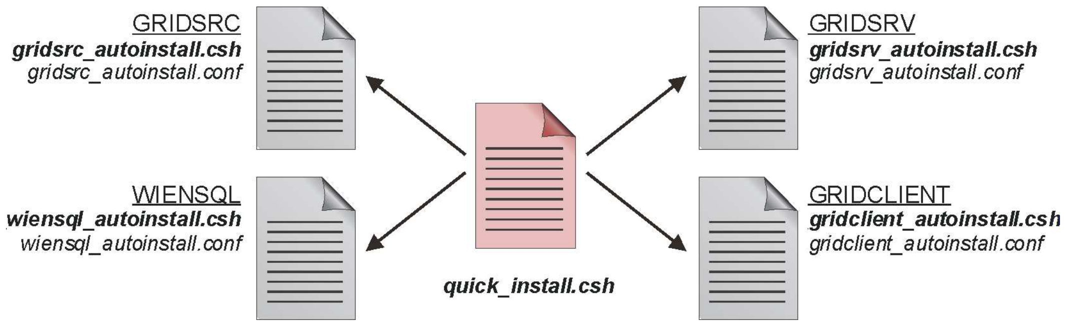


Figure 3.3: Four individual scripts are subsequently invoked by the script ***quick_install.csh*** in the course of a default installation

*>**quick_install.csh LAYER(S)***

, where the argument 'LAYER(S)' is one of the layers of Fig.3.2 and may be a string out of the following list, whose individual items have already been discussed in the previous sections.

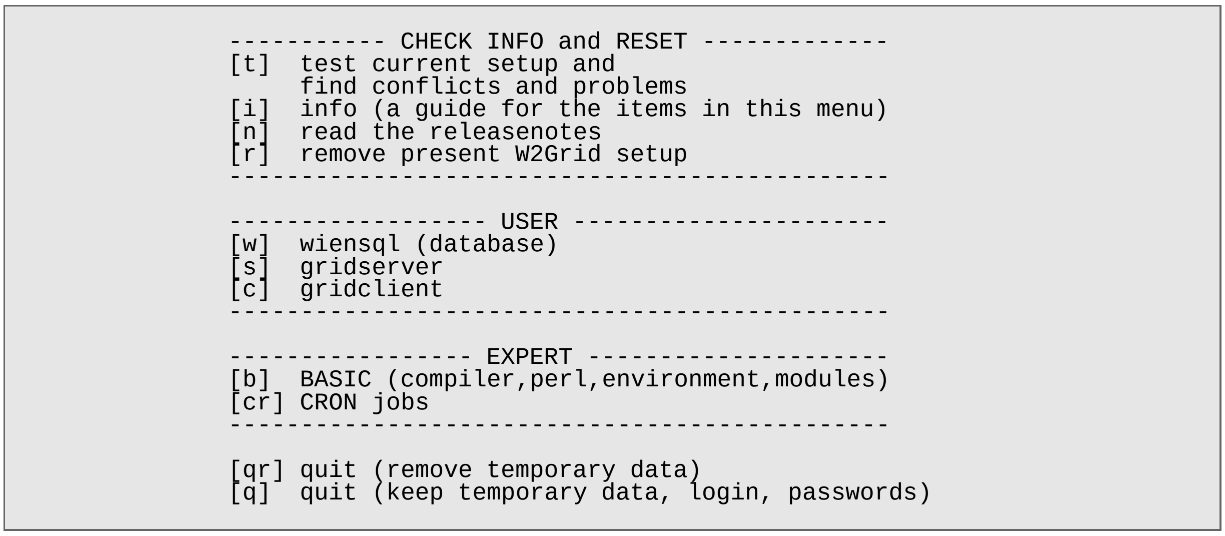
- **GRIDSRC:** see section 3.1.1.1
- **WIENSQL:** performs all GRIDSRC-tasks + Installs the database
- **GRIDSRV:** performs all WIENSQL tasks + installs only the GridServer
- **GRIDCLIENT:** performs all WIENSQL tasks + installs only the GridClient
- **GRIDSRV,GRIDCLIENT:** (or the other way round) will install both, the GridServer and the GridClient as well.

The individual scripts shown in Fig.3.3 are also accessible independently. Due to the hierarchical order of the layers and the way, the master-script interprets the input, it is not possible to run the program ***quick_install.csh*** twice on the same host without a prior removal of the previous installation, hence 'adding' a layer (e.g. the GridClient) afterwards is not possible with the master-script. For this purpose the manual installer (section 3.1.3) is provided.
The quick-installation is usually recommended to perform a default-setup of the **W2GRID** components, since the settings of the config-files will suite most hosts, whereas the selection of

the proper plugins and their configuration is to be done manually afterwards anyway. The installation scripts illustrated in Fig.3.3 glue together a greater set of very basic tools (Perl and csh), some of which are also used by the menu-guided installation (see Fig.3.5).

3.1.3 Menu-guided (interactive or manual) installation and configuration

In addition to the capabilites, which are also provided by ***quick_install.csh***, the manual installation offers tools to select and configure plugins, install/remove packages, start/stop the daemons and to modify each part of the installation. It is possible to remove parts of it or install additional ones. A sample screenshot of the main-menu (***bin/gridsrc_install.csh***) is shown in Fig.3.4. The master-script is launched by the command



```
----------- CHECK INFO and RESET -------------
[t]  test current setup and
     find conflicts and problems
[i]  info (a guide for the items in this menu)
[n]  read the releasenotes
[r]  remove present W2Grid setup
---------------------------------------------

------------------ USER ----------------------
[w]  wiensql (database)
[s]  gridserver
[c]  gridclient
---------------------------------------------

----------------- EXPERT ---------------------
[b]  BASIC (compiler,perl,environment,modules)
[cr] CRON jobs
---------------------------------------------

[qr] quit (remove temporary data)
[q]  quit (keep temporary data, login, passwords)
```

Figure 3.4: Screenshot of the startup screen of the manual installer

$>$***cd $GRIDSRC/bin***[8]***;gridsrc_install.csh***

Both methods (menu-guided and quick-install), however, are basically only frameworks that glue together several Perl- and csh-scripts, which are used from within both. This is illustrated in Fig.3.5. The main difference between ***quick_install.csh*** and ***bin/gridsrc_install.csh*** (apart from the greater capabilities of the latter one) is the source of the input data. In the case of the menu-guided installation the input is requested at the commandline each time a tool is invoked, whereas in the case of the script-based installation, the input is read once, at startup, from the config-file and then supplied to the tool.

[8]Although ♯bin/ is added to the $path variable of the shell, the script has to be executed from within this directory.

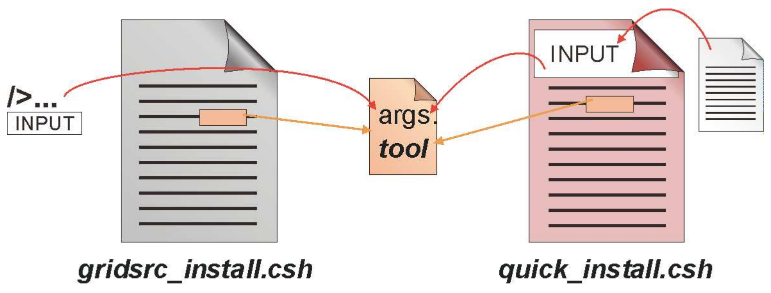


Figure 3.5: The installation scripts ***gridsrc_install.csh*** and ***quick_install.csh*** employ the same tools, only their input is acquired differently

3.1.4 Removal

The command

>***reset.csh***

removes **W2GRID** completely. The shell-script runs the layers of the infrastructure in the reverse order and first stops all processes, afterwards removes the daemons, then the database and finally deletes the temporary directories and cleans the **W2GRID**-specific entries from the startup-files.
Partial removals of individual components (e.g. the GridServer) without removing **W2GRID** completely is also possible, since each of the sub-menus provides an option **[r]** for removing the respective component (see Fig.3.4). If changes have been made to the environment variables, it is recommended to reload the startup-files (i.e. the session) afterwards. The necessity of doing so will be announced explicitly by the master-script before quitting.

3.2 Usage

3.2.1 Daemon processes

After installing the infrastructure, it may be necessary to start/stop the daemons or to obtain additional verbose output for debugging during the development of **{COMMANDS}** and ⌜*JOBS*⌝.
A simple way to start and stop daemons is accessible from within the interactive installer script ***gridsrc_install.csh***, whereas other operations (e.g. verbose output) require to work on the commandline. To change the behaviour of a daemon manually, soft-links to all three daemons can be found in the directory ♯demons/[9]. The available default-options are: **start|status|restart|stop**.

[9]The misspelled name has been defined at an early stage of development and kept for the names of this directory, some files and functions in order to avoid conflcts, although the correct spelling would be 'daemon'.

Exactly one argument has to be supplied (e.g.)

>***demons/gridsrvd start***

The Perl-daemons additionally accept the following arguments:

- **--verbose [ALL|SOCKET|STANDARD|...]:** Allows to obtain (selected) debug output. For a complete list of all options use the flag '-h'
- **-nojobs:** Prevents the original process 1 $_{2.14}$ from forking into the daemon 2 $_{2.14}$ and controller 4 $_{2.14}$. Only the RPC-commands (section 2.6.2) are available, whereas the $\ulcorner JOBS \urcorner$ (section 2.6.3) are disabled.
- **-keepterm:** Will not set the process into the background and hence not return the terminal to the user. Terminating the process with CTRL+C will shut down the daemon safely.
- **-nocrypt:** Does not use the AES encryption and forces the server as well as the client to send and receive data in plain text. This may be helpful to improve the performance or to debug the logon-protocol.

Additional options and flags are explained in the usersguide [97].

3.2.2 Status check

A quick status check can be obtained by the use of the csh-script ***gridstatus.csh***. A sample output is provided in Fig.3.6.

```
checking authentication ... done
W2GRID INSTALLED     : YES
WIENSQL INSTALLED    : YES
WIENSQL RUNNING      : YES
GRIDSERVER INSTALLED : YES
GRIDSERVER RUNNING   : YES
GRIDCLIENT INSTALLED : YES
GRIDCLIENT RUNNING   : YES
```

Figure 3.6: A sample output of the C-Shell script ***gridstatus.csh***

3.2.3 Commandline interfaces

`W2GRID` offers three Perl commandline interfaces (one for each daemon), which are described in the following sections. The most important flags and options of these commandline-tools are itemized as follows:

- **-posix** Includes the posix-library, which is needed to catch single key-strokes like the arrow-keys and thereby allows to keep a 'history' of the entered command strings in the same convenient way as most common terminals.
- **-S [COMMAND]** Runs an command (e.g. an SQL query) in batch mode and closes the connection immediately afterwards.
- **--verbose [MODE]** The argument supplied to this option is a string out of 'STANDARD, ERROR, WARNING, SOCKET, INTERNAL, WIENSQL, ALL' or a combination thereof (separated by colons).
- **-h** Shows a complete list of available options and flags.

Some arguments are only available for the commandline interfaces of the two Perl-daemons.

- **-progress** displays the remote progress of the currently processed command in %.
- **-time** The first line of output will show the time-difference in seconds between the submission of the command to the daemon and the arrival of the corresponding result.

3.2.3.1 GridServer interface (*gridsrv_console.pl*)

Allows to contact the local GridServer directly. It reads the connection data (**(host_port)**, **(host_key)**) from the registry and employs the default connection plugin to contact the daemon at the local IP address. It is supposed to be used by developers for debugging server-side **{commands}** and ⌜*JOBS*⌟. Most of its input and output is based on XML strings for better machine-readability. It is located in ♯SRC_gridsrv/bin/ and invoked from the commandline by:

>***gridsrv_console.pl [FLAGS] [OPTIONS]***

For sample commands it is referred to the usersguide [97].

3.2.3.2 GridClient interface (*gridclient.pl*)

It is provided for the regular user, hence the input as well as the output is plain text. It is located in ♯SRC_gridclient/bin/ and invoked with:

>***gridclient.pl [FLAGS] [OPTIONS]***

Some sample commands are explained in section 3.2.4.

3.2.3.3 Wiensql interface (*wiensql.pl*)

The commandline interface to the wiensql-database is located in ♯SRC_wiensql/bin/. It is invoked with:

>***wiensql.pl [FLAGS] [OPTIONS]***

Only the most basic syntactical elements of ANSI SQL have been implemented in a non-relational way, hence its capabilities are very limited but sufficient for the all-day-use of **W2GRID** and its components.

element	***example***	***comment***
SELECT	*select * from jobs.server*	no natural joins
INSERT	*insert into jobs.server (***) values (***)*	
UPDATE	*update jobs.server set ***=*** where ****	
CREATE	*create table jobs.server (***)*	
	create user 'login' ('pwd','db','name','etc.')	
	create database 'db'	
DELETE	*delete * from jobs.server where job_id=****	
	*delete user ****	
	*delete table ****	
	*delete database ****	
WHERE	*...where job_id=****	and,or,<,>,>=,<=,like
LIKE	*...like '%job%'*	only for text-columns

Table 3.1: Syntactical elements of wiensql

In addition to the default arguments ***wiensql.pl*** supports also the following ones:

- **-o [TYPE]** The output is by default formatted as a tabular frame. It may also be XML-wrapped or a frame less list, according to the supplied [TYPE], which is a number from 0-5 (the default is 0).
- **-f [FILE]** Reads the SQL instructions from a file (e.g. used for creating the tables during the installation process).
- **-host [HOST], -port [PORT], -key [KEY]** By default the interface connects to the local daemon and extracts the required connection data from the file $WIENSQL_root/.wiensqlrc[†]. If supplied different values, the interface can also contact remote wiensql-daemons.
- **-l_sql [LOGIN]** Specify a different login than the default 'anonymous' (e.g. gridsrv_$hostname)
- **-C [COMMAND]** Interprets non-SQL commands (e.g. 'shutdown', 'ping' or 'dump') in batch mode.

3.2.4 Selected GridClient-commands

This section illustrates the usage, purpose and output of some of the GridClient default-commands, which are available already without any additional application plugin. The respective GridServer commands are omitted, since this would exceed the scope of this document. The interested developer is referred to the usersguide [97].

3.2.4.1 Built-in commands

Several commands are interpreted directly by the parent process of the daemon and bypass all the complicated child processes illustrated in Fig.2.17. The advantage is a significantly shorter response-time, since they do not have to include their workflow from a file.

- **{ping}**$_C$ returns '1'. A 'ping' internally precedes each command to check if the connection is open.
- **{?}**$_C$ or **{help}**$_C$ displays a list of all available commands.
- **{shutdown}**$_C$ stops the daemon.
- **{version}**$_C$ displays the daemon version
- **{exit}**$_C$ or **{quit}**$_C$ terminates the connection.

3.2.4.2 Pathnames

Commands, which need a path as input, may also get the current path from the interpreter. **{pwd}**$_C$ is used to display this path, **{cd}**$_C$ for changing it and **{dir}**$_C$ for showing the content of the directory. This internal handling of path names is especially convenient if the commandline interface is used in batch mode from within a script.

3.2.4.3 Hello world

The 'test' workflow **{test}**$_C$ is the **W2GRID** version of a “Hello World” program and returns only a single string (see the source code in Fig.A.4 in the appendix). It is provided for debugging and serves as a template.

3.2.4.4 GridServers and their properties

The host-registry has been explained in section 2.6.6. This table contains informations about the registered GridServers and the plugin-configuration data for connecting and transferring files.

- **{host.list}**$_C$ retrieves all hosts from the registry and displays them. The output may be customised as shown in Fig.3.7.
- **{host.check}**$_C$ forces the check, which is otherwise run by the job $\ulcorner$*host.check*$\lrcorner_C$ automatically once a day in the background to be performed immediately.
- **{host.info}**$_C$ provides extended informations about specific hosts. Either the **(host_name)**, the **(host_id)** or **(host_ip)** must be supplied as a mandatory argument.
- **{host.usage}**$_C$ returns a report of the current resource consumption as reported from all registered hosts. The results are presented in a uniform (queuing system independent) format.
- **{host.add}**$_C$ adds a new GridServer to the registry, which can be modified with **{host.modify}**$_C$ and removed with **{host.delete}**$_C$. It is recommended to use the corresponding and more convenient (interactive) functionalities of the menu-guided installer ***gridsrc_install.csh*** instead.

```
>host.list --format "[ID %id]  %n   %nds nodes   %c (%ip:%p)" -c
#command took 2 seconds to complete
[ID 2] athena 16 nodes ONLINE (128.130.134.045:8888)
[ID 3] aurora 72 nodes ONLINE (128.130.033.145:8334)
[ID 4] gescher 16 nodes ONLINE (131.130.186.180:8180)
```

Figure 3.7: The result of **{host.list}**$_C$ with a user-defi ned formatting

3.2.4.5 Show/update/delete resource slots

The persistent storage containers have been explained in section 2.6.5.

- **{slot.create}**$_C$ creates a new slot and returns the respective **(slot_id)**$_c$.
- **{slot.list}**$_C$ displays a summary of all currently existing slots.
- **{slot.kill}**$_C$ removes an entry from the slot-registry e.g. after a calculation is finished successfully. It needs the **(slot_id)**$_c$ as a mandatory argument.

3.2.4.6 Show/update/delete jobs

Background-tasks have been explained in section 2.6.3. These $\ulcorner$*JOBS*$\lrcorner$ avoid the 'blocking' effect of RPC-commands on the one hand and perform regular tasks in the background on the other hand.

- **{job.list}**$_C$ displays the contents of the job-registry.
- **{job.kill}**$_C$ removes an entry. If the respective process is executed at the moment, just its registry-entry is deleted, but the process is not stopped.
- **{job.nextexec}**$_C$ displays three dates/times. The current date, the last run of the controller 4 $_{2.14}$ and its next run.
- **{job.run}**$_C$ bypasses the controller and immediately executes a $\ulcorner JOB \urcorner$ (still in the background).

3.3 Development

The concept of core and plugins, which is explained in section 2.9 simplifies the way, how developers can contribute their own code to **W2GRID**. Knowing Perl is the only requirement, however it is possible to reduce the amount of Perl-code to an absolute minimum, which only glues together external components (see section 3.3.5.2). The components, which may be contributed by third-party developers are:

- (RPC-) commands (see section 2.6.2)
- Background tasks $\ulcorner JOBS \urcorner$ (see section 2.6.3)
- Platform plugins for interoperability with additional operating systems (see section 2.9.1)
- Execution plugins to submit/control/steer tasks by the use of a queuing system (see section 2.9.2)
- Filetransfer plugins (see section 2.9.4)
- Connection plugins (see section 2.9.4).
- Processor plugins, which supply the arbitrary performance numbers for a given CPU (see section 2.9.6)
- Application plugins for integrating third-party software (e.g. HPC programs) into **W2GRID**. Apart from scientific applications, this kind of plugin also allows to introduce e.g. support for other databases than wiensql (see section 2.9.5).

Templates and numerous sample scripts are provided in the appendix.

3.3.1 Commands

Commands are an essential ingredient for most application plugins and may also serve for debugging and testing basically any part of **W2GRID**. The purpose and method of their operation is explained in section 2.6.2, whereas this section focuses on the development of the respective source code. A typical file structure as illustrated in Fig.3.8 is not mandatory but recommended. The line numbers used in the following explanation refer to the sample code provided in the appendix (Fig.A.4). All initial settings such as loading libraries (line 7) or defin-

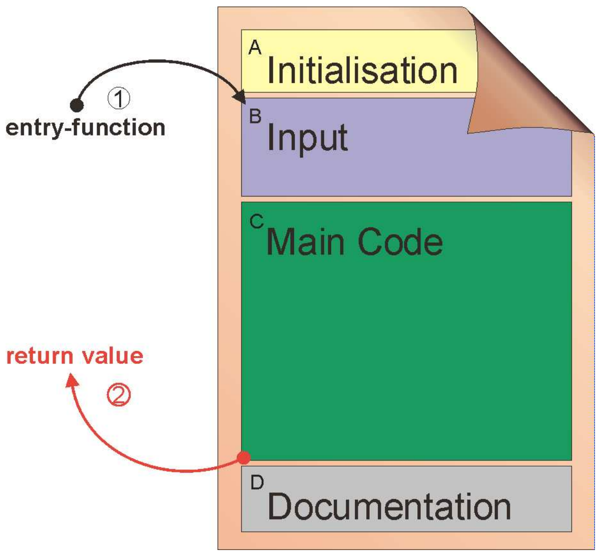


Figure 3.8: Sections of a RPC-command fi le

ing/setting global variables is performed in the first section A, whose code is not contained in any function and hence executed right after including the file and even before the entry function &exec_request() 1 (line 12) is called. The sequence of tasks executed afterwards is up to the developer, it is however recommended to start with the processing of the arguments B (lines 19 - 24). The help-flag, which should be supported in any command (line 19) obtains its output from a subroutine (lines 55 - 70), which is part of section D. The name of this (optional) subroutine is up to the developer. At the end of section B it is recommended to check for invalid arguments (line 24, see also section 6).

Section C contains the workflow-code. Since it should only be executed if no error has occurred yet, it has to be enveloped in a condition (lines 30 - 33). The given sample just returns a string, but section C may contain virtually anything. At its end, the result must be written into the result-buffer (line 32), otherwise the RPC-command will return an empty string. The function is left with the 'return' statement in line 39. The formatting of the return-value 2 should be left to the provided function, otherwise the string, which is directly sent to the client may not

be readable.
A second entry-function &exec_help() is defined, which returns a single-line documentation if the command has been submitted with a preceding question mark (e.g. **{?test}**$_{S,C}$). The function (lines 42 - 49) is part of the documentation D and is usually not invoked from within the other parts since it uses the same exit-function as section C to format the result-string.

3.3.2 Jobs

⌜*JOBS*⌟ are used for background tasks. They are not invoked directly by the user from the commandline[10]. Instead they are mostly run by the controlling process 4 $_{2.14}$ in regular intervals. The purpose and their workflow is explained in section 2.6.3. The source code (Fig.3.9) of a

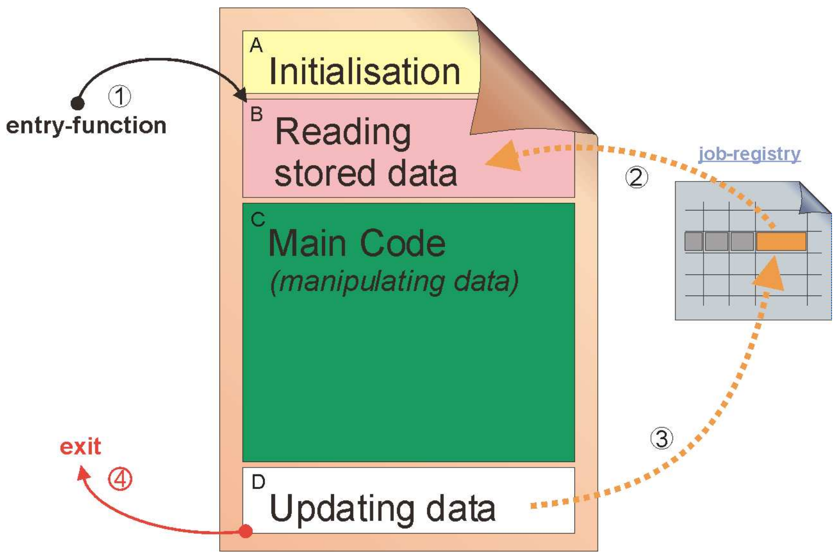


Figure 3.9: Sections of a ⌜*JOB*⌟-fi le

⌜*JOB*⌟-file contains four important sections, which are not mandatory but recommended. They are explained in the the following paragraph, whose line numbers refer to the sample Fig.A.5 in the appendix.
Upon including the file, the initialisation A is invoked instantly in the same way as before (section 3.3.1), since it is not enveloped by any subroutine. Its exact position is irrelevant, the respective code (e.g. including additional libraries, defining global variables) may therefore also be placed at the end of the file, although it is recommended to insert it into the provided and commented section at line 7.
The entry-function &exec_command() 1 starts the ⌜*JOB*⌟-workflow and contains all relevant code lines 13 - 47, such as the processing of the stored ⌜*JOB*⌟-data, which are read from the

[10] although it is possible to force the immediate execution of a ⌜*JOB*⌟ by the aid of **{job.run}**$_{S,C}$ **(job_id)**

database 2 and stored in a buffer in the background. The items of interest may be extracted on demand (section B, lines 19-20 -). The main-part C processes the workflow of the ⌜JOB⌟-script. Because no output can be sent to the client, all data must either be written to files (e.g. logfiles) or to the job-registry, which is not done immediately. Instead all updates are accumulated until section C is finished and finally 'committed' to be written to the table 3 in the final section D, which concludes the function &exec_command(). The ⌜JOB⌟ exits without return-value 4. Different to commands, a documentation does not make sense because it cannot be displayed anywhere. The developers are however encouraged to attach comments to their code.

3.3.3 Important libraries

The **W2GRID** code spans roughly 150.000 lines and almost 1500 unique functions. The majority of them is used only internally and their explanation clearly exceeds the scope of this thesis, hence the reader is referred to the developersguide [103]. Yet for completeness and to illustrate the capabilities, a list of frequently used libraries and their purpose are presented in this section, whereas the most important functions thereof are explained in the appendix (section 6).

- **Utilities**: Basic I/O such as queries and formatting of data, encryption/decryption, creation of temporary directories and files, extraction of data from files and strings, file-statistics.

- **Exception handling**: The provided functions offer methods to write, read and check for error-messages and warnings, which are automatically appended to any result-data returned by the daemon. Thus the developer does not have to handle them explicitly.

- **Verbose and regular output**: The verbose-strings may be used to debug scripts in their development phase. To see them on STDOUT, the verbosity of the daemon must be enabled (section 3.2.1). An additional benefit of the verbose-commands is their use for an intrinsic documentation of the Perl-scripts.

- **Logfiles**: **W2GRID** offers to capture all output of warnings, errors and verbose-strings in logfiles, which is a convenient way of tracking errors, even if the verbose output has not been enabled. All the developer needs to do is opening a logfile and placing several verbose-statements at important sections of the code to e.g. preserve the content of certain variables.

- **Execution plugin**: Its functions provide the necessary capabilities to attach to a given queuing system and submit jobs, check their state as well as the state of the overall queuing system or cancel them. The same function names, their input and output is identical in every library, therefore an application plugin must use these functions instead of hard coded calls to any explicit queuing system. The important functions are discussed in section 3.3.7.

- **Slots**: The functions allow to create slots, manipulate their content and delete them. The content of their **(parameter)** column makes use of XML-datagrams, which require to use functions of an additional library (see below).

- **Jobs**: This library allows to create, manipulate and delete background tasks(⌜*JOBS*⌟). A special feature about the respective functions is the fact, that all ⌜*JOB*⌟ related data is retrieved from the table and already processed before the entry-function of the script is called. Therefore all data is already available and does not have to be fetched and processed explicitly by the developer. If changes are made to the data, all respective functions of this library do not immediately write the new content to the table. Instead it is stored in a temporary container and committed in one of the last lines of the script before it quits. If this function is omitted for some reason (e.g. a crash happens above this line), all data collected up to the crash is lost.

- **Misc. daemon functions**: Provided to read and manipulate the content of the registry.

- **GridServer connection**: This library offers the functions, which interact directly with a GridServer. The opening of a connection, the submission of commands by the use of the respective protocol, the transfer of files (put and get), the termination of the connection.

- **Files**: Files[11] are handled by **`W2GRID`** in an object-like manner. A hash, referred to as the %filelist, contains individual ’objects’ %filelist{0...COUNT-1}. Each of them is again a hash and contains items like the NAME, the PATH and SIZE of a file. The predominant purpose of the filelist is to collect and handle a large array of filenames with a single object and by template-functions, which allow the developer to perform some common file-operations: Copying from source to target, packing into an archive, concatenating the names into a long string to be supplied to any tool (such as ***tar***), updating the properties (e.g. the SIZE), applying modifications (e.g. replacing the template-string ’CASE’ by the actual name ’tic’ simultaneously in all stored filenames).

- **XML-datagrams**: Datagrams are frequently used in **`W2GRID`** for data transfer and to wrap complex parameters or results. This is especially essential for storing complex data such as the numerous configuration-data in the registry or turning a %filelist, which

[11]or more precisely the ’fi lenames’ and the important properties of the same. Basically everything but the actual fi le content.

is basically a specifically formatted segment of memory into something, which can be stored in the e.g. slot-registry. A sample code in the appendix (Fig.A.10) illustrates their use.

- **wiensql**: A very important feature is the database-connectivity, which has to be available to all commands and $\ulcorner \mathit{JOBS} \urcorner$. Most of this file's content is irrelevant to the developer, since the usual operations performed on certain registry-tables can be done more efficiently by proper slot or $\ulcorner \mathit{JOB} \urcorner$ related commands, which also care for the formatting of the result. In other cases, the wiensql-functions have to be used. They allow to select data either as a single row or a complete record-set, update, delete and insert data as well as open and close the database-connection.

- **Input**: Arguments may be supplied to tools and commands[12]. Hence it is necessary to process these strings and extract the desired informations.

- **Shared utilities**: The processes covered by this libraries are quite complex, but very important for application plugins. They allow to perform a filetransfer by simply supplying the %filelist variable as a reference. This single statement is sufficient and saves the developer many thousand lines of code. Another function determines the state of a calculation based on the combined states of the respective slot and remote $\ulcorner \mathit{JOB} \urcorner$. The third crucial function is used to run parallel processes.

3.3.4 Important variables

`W2GRID` offers several variables, which are available by default in all scripts. They are defined in the library file $GRIDROOT/libs/global.pl†. The most important are:

- **$PATH{LIB}** ♯$GRIDSRC/libs_perl/
- **$PATH{SHAREDLIB}** ♯$GRIDSRC/libs_perl/gridshared/
- **$PATH{SYSTEMLIB}** ♯$GRIDROOT/libs/
- **$PATH{SRVLIB}** ♯$GRIDSRC/libs_perl/gridsrv/
- **$PATH{CLIENTLIB}** ♯$GRIDSRC/libs_perl/gridclient/
- **$PATH{LOG}** ♯$GRIDROOT/log/
- **$PATH{BIN}** ♯$GRIDSRC/bin/

[12] $\ulcorner \mathit{JOBS} \urcorner$ in the contrary use other methods to retrieve input (see above).

- **$PATH{TEMP}** will point to a temporary-directory assigned to each command/⌜*JOB*⌟, that will be destroyed once the command/⌜*JOB*⌟ is done. Storing essential informations, which shall outlast the scope of a command/⌜*JOB*⌟ is therefore not recommended. The directory will also be removed if a ⌜*JOB*⌟ is run subsequently, because it is bound to the PID. Since the child process has a different PID, it is not possible to keep the directory.
- **$PATH{CURRENT}** The current path, which is passed by the client.
- **$GLOBAL{HOSTNAME}** The local hostname.
- **$GLOBAL{LOGNAME}** The login of the local user.

3.3.5 Application plugin

In order to integrate an application into **`W2GRID`**, it is necessary to write the corresponding RPC-commands and ⌜*JOBS*⌟ for the GridServer and the GridClient. Certain tasks, which have to be performed by both daemons may be put into libraries, which are recommended to be placed into the directory ♯libs_perl/$app-name/. General tools should be provided in ♯bin/$app-name/ or simply ♯bin/. For the implementation of complex workflows exist basically two different approaches, which are illustrated in Fig.3.10 for comparison.

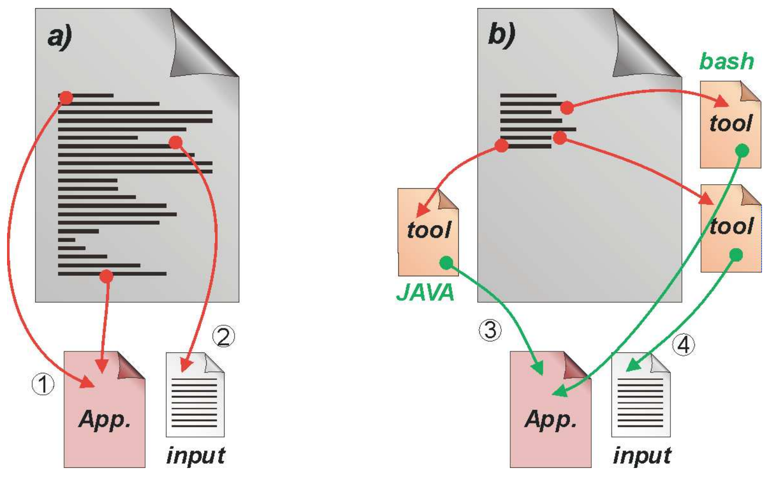


Figure 3.10: Two different approaches for implementing application plugins.

3.3.5.1 Full implementation **a** 3.10

If the developer is familiar with Perl and if the plugin requires numerous regular expression operations it is recommended to code all instructions required for wrapping the application in the command-script, since it is a very convenient language to handle regular expressions, and most of the required tools are already provided as templates too. The resulting plugin is therefore fully portable and does not depend on external tools other than provided by the application itself. The only disadvantage is, that the use of Perl is mandatory. If some external tools exist already but are written in any other language, they will have to be re-written. Usual instructions, which are part of this approach invoke/steer the application directly 1 or write/modify some input files 2.

3.3.5.2 Partial implementation b 3.10

If there are already some existing scripts and tools, they don't have to be re-written. Instead, the Perl-code of the commands and ⌜*JOBS*⌟ may be used to glue together the individual components, whereas the steering 3 of the application or the modification of certain files 4 is done by the external tools and not directly from within the plugin. This is especially useful, if the developer is not familiar with Perl or prefers other languages such as Java, Python or Fortran to write the workflows. Whereas this may be a convenient approach for some applications, it comes with the disadvantage to introduce external dependencies, which can limit the portability.

3.3.5.3 Sharing the plugin

It is recommended to write proper installation and removal instructions and to share it as a **W2GRID**-package (see section 3.3.11). In the course of installing such a package, the necessary paths will be created automatically (depending on the instructions) and the commands, ⌜*JOBS*⌟, library files and tools copied to their proper location. After some optional database-operations, the plugin is operable without having to restart any of the daemons.

3.3.6 Platform plugin

This plugin is included in every Perl-script and will interact with the operating system. It needs a single library file, which is recommended to be stored in ♯libs_perl/platforms/. Upon installation, it is copied to ♯$GRIDROOT/libs/ and renamed to system.pl†. For development, it is recommended to take an existing plugin as template (e.g. redhat.pl†), copy it to the new target system (e.g. knoppix.pl†) and replace the corresponding functions, whose names, input- and output-format are mandatory.

- **&exists_pid(*$PID*)**: Checks, if the $PID still exists in the process-table and returns '1' or '0' correspondingly.

- **&process_memory(*$PID*)**: Returns the amount of memory (MB), which is occupied by the process $PID.
- **&nslookup(*$host*)**: Returns a hash, containing the HOSTNAME and the IP. It usually employs different tools such as 'dig', 'host', 'arp', etc... to obtain the result.
- **&childprocesses(*$PID*)**: Returns an array containing the list of child processes, which belong to $PID.
- **&getload()**: Returns the current processor load. Usually &exec__load() is used instead.
- **&read_local_ip()**: Returns the local IP address.
- **&totalmem()**: Returns the physical RAM.
- **&freemem()**: Returns the free RAM.
- **&list_processes()**: Returns a hash containing all PID's currently present on the system (ps -ef). The number of processes is returned as $RESULT{COUNT}. The individual PID's are accessible as $RESULT{0}, $RESULT{1}, etc...
- **&sysps(*$string*)**: Takes the output of a 'ps -ef' call (only a single line) as an input and returns the PID.
- **&syspsf(*$string*)**: Same argument as &sysps() but it returns a hash, which contains the elements OWNER, PID and PARENT.
- **&systop()**: returns the output of a 'top' command. The result is a hash, containing the COUNT-element to indicate the number of its entry. Each entry (e.g. $TOP{0}) is a hash, too. It contains the elements **RAW** (rawtext, the complete line), **PID**, **USR**, **VIRT** (virtual memory), **SHR** (shared memory), **CPU** (processor usage in %), **MEM** (%), **TIME** (unformatted), **CMD** (command).
- **&getuptime()**: Returns the uptime.
- **&current_dir()**: Returns the current directory (absolute path).
- **&get_home()**: Returns the home-directory $HOME or ~/ (absolute path).
- **&read_time(*$string)***: String-to-date conversion. Each host uses a defined locale date- and time-format, which has to be converted frequently to the format used internally by wiensql (HH:MM:SS dd-mm-yyyy).

This plugin has to be registered to the plugin-registry, before it can be used. For this purpose, an XML-descriptor is to be provided in the directory $^\sharp$plugins/server/platforms/. A sample file is shown in the appendix (see Fig.A.11). It is recommended to name the new descriptor after the platform name, too (e.g. knoppix.plg†). The most important value to be replaced is the <ID>-tag in line 2, because this is used for identifying the plugin. The path to the newly created library file has to be supplied in line 16 and is mandatory, too as well as the endian-type (little '0' or big '1') in line 20. To use the plugin, it is sufficient to copy both new files (the library file containing the functions and the XML-descriptor) to the specified locations on the target machine and to run

$>$***reinstall_plugins.csh***

which updates the plugin-registry. Afterwards, it can be installed and configured by the use of the manual installer (see above).

3.3.7 Execution plugin

This is the most important yet most complex plugin of **W2GRID**, since it allows applications to be run on a given infrastructure. The execution plugin is included into every GridServer command and $\ulcorner JOB \urcorner$ by default and does not have to be loaded explicitly. The GridClient in the contrary does not need it. Upon installation, it is copied to $^\sharp$\$GRIDROOT/libs/ and renamed to exec.pl†. The function names are mandatory, too and have to be adjusted to the queuing system. It consists of a library file, usually stored in $^\sharp$libs_perl/execs/ and the mandatory plugin descriptor. A sample XML-file for PBS is provided in the appendix (Fig.A.15). Its important functions are:

- **&exec__submit(*\$dir,\$command,\%parameter*)**: To submit jobs, the base-directory \$dir, where the command shall be started and the \$command itself are the two mandatory arguments. The command must not contain any MPI or other instructions, since these details are supposed to be added by **W2GRID** according to the previous setup of the plugin. The parameter-list (HASH) contains several additional informations about e.g. the resources (number of nodes, expected runtime, required memory). A sample code is provided in the appendix (see Fig.A.7, After submission, the function returns the internal ***PID***$_Q$, whose meaning and format is only 'known' to the queuing system.
- **&exec__state(*\$PID*$_Q$)**: checks, if the \$***PID***$_Q$ is currently executed or queued. Returns '-1', if the function produced an error, '0' if the \$***PID***$_Q$ does not exist, '1' if it is queued and '2' if it is running. In any other case (e.g. stopped) it will return '3'.

- **&exec__kill(*$PID*$_Q$)**: Removes the process $***PID***$_Q$. Returns '-1' in the case of an error '0' if the kill has not been submitted successfully and '1' if the job is gone.
- **&exec__refresh()**: Refreshes the informations about the state of the queuing system (running jobs, available nodes, etc.)
- **&exec__load()**: The ratio of busy-nodes / total-nodes in %.
- **&exec__dynamic()**: Reveals, if the name of the nodes may be known in advance (single host or master of a virtual **W2GRID**-cluster) or if the queuing system will supply the nodes after the submission-script is started.
- **&exec__queued()**: Number of queued jobs.
- **&exec__starttime(*$cpus*)**: An estimated starttime in seconds, specifying the time when the given number of $cpus might finally be available.
- **&exec__forecast(*$processes*)**: The fraction of CPU capacity for one of the requested number of $processes[13]
- **&exec__freenodes()**: the number of free nodes.
- **&exec__nodes()**: The total number of nodes
- **&exec__maxnodes()**: The maximum number of nodes as supplied during the configuration of the plugin.
- **&exec__power()**: The product of CPUs per node and cores per CPU. A two-processor node with dual-core CPUs will return '4' as a result.
- **&exec__totalmem()**: The total memory of a single node (or total in the case of shared-memory). This value is taken from the configuration.
- **&exec__freemem()**: The free memory on a single node. Use &exec__isshm() to check if this is individual or total-memory.
- **&exec__existspid(*$PID*$_Q$)**: checks if the $***PID***$_Q$ still exists on the queuing system. Returns '1' or '0'.
- **&exec__initialize()**: Initialises the plugin (e.g. loads current jobs and the system-status and fetches the configuration-data from the database).

[13]e.g. If a dual-core processor already executes a single and shall serve an additional one, each of the two will get 100%. If two tasks run on a single-core CPU, each will get only 50% (assuming the same nice-level).

- **&exec__ismpi()**: Checks, if MPI has been enabled during the configuration. Returns ’1’ or ’0’.
- **&exec__isshm()**: Checks whether the memory is shared or not. Returns ’1’ or ’0’.
- **&exec__version()**: Returns the version number. This is critical, since it allows an application to determine, which capabilities are supported by the installed execution plugin and to judge if the host can be used at all. The numbers and the capabilities (respectively the functions) must correspond to the given definitions in the developersguide [103]. At present all execution-plugins are available as version 1. This feature has been introduced to avoid incompatibilities in the future as different capabilities are indicated by different numbers.
- **&exec__isthread()**: Checks if the threading has been configured. Returns ’1’ or ’0’.
- **&exec__probability(*$cpus*)**: Returns an estimated probability in % for the chance that the requested number of $cpus are available at once. In the case of a managed cluster this should usually return 100% assuming that the respective queue exists.
- **&exec__jobs()**: Fetches the statistics of running and queued jobs and presents a summary, which is used for the command **{host.usage}**$_C$.

The plugin has to provide the respective library file and the XML descriptor. Optionally it may also come with some commands to test the functionality (e.g. **{pbs.test}**$_S$). The command **{test.exec}**$_S$ may be used as a default command to test those functions, which return simple strings or numbers. It uses the currently installed execution plugin, but has to work for all different ones.

3.3.8 Connection plugin

It requires a library file, which should be usable by the GridClient and the GridServer. It is stored in the directory ♯libs_perl/connections/ and different to the platform and the execution plugin, it is not copied anywhere upon installation. The XML plugin-descriptor has to be different for each of the two Perl-daemons.
Whereas the functions provided by the platform plugin and the execution plugin may be used directly in the code of commands and $\ulcorner JOBS \lrcorner$, the functions of the filetransfer plugin (and the connection plugin, too) are called indirectly. This is referred to as ’mapping’ and it is necessary, since there may be several different filetransfer plugins loaded at the same time to serve different connection methods to each individual GridServer. For this reason it is obliging to use

different function names for each of these libraries[14]. The following functions are presented by their purpose and the name of the corresponding tag in the XML plugin-descriptor (see Fig.A.14 in the appendix):

- **&connection_init(\*%data,$path*)**: Initialises a connection (lines 46 - 49). The hash contains the data, which have to be supplied upon installation and configuration of this plugin (e.g. IP, port, etc.). The $path indicates the location, where the progress-packages (see section 2.6.11) have to be written to. It is not mandatory but strongly recommended to support this feature, otherwise the progress indication will not work. The initialisation spans all tasks, which have to be performed only once <u>before</u> a connection is opened.
- **&connection_open(*$string*)**: Opens the connection. It must be possible to call this function again after closing the connection, without having to initialising it again (lines 21 - 23).
- **&connection_close()**: Closes the connection (lines 41 - 43).
- **&connection_status()**: Returns either '1' if the connection is still open or '0' if it is already closed (lines 36 - 38).
- **&connection_read()**: Reads data from the connection (lines 31 - 33).
- **&connection_write(*$string*)**: Writes the $string into it (lines 26 - 29).

An illustration of the 'function-mapping' is provided in Fig.3.11. The template function name,

```
1:<ITEM>
2:    <NAME>connection_open</NAME>
3:    <DESCRIPTION>Routine for reading data</DESCRIPTION>
4:    <VALUE>standard_socket_open</VALUE>
5:</ITEM>
```

Figure 3.11: The 'function-mapping' of the connection plugin

which is used as a variable internally by **`W2GRID`** is provided with the <NAME> tag (line 2) (e.g. 'connection_open') and must not be changed. The actual function name, the template is to be mapped to, has to be supplied with the <VALUE> tag (line 4) (e.g. 'standard_socket__open').

3.3.9 File transfer plugin 'ftp'

The development is similar to the previous one and requires a single library file (or an individual one for each daemon), which contains all elemental functions for the filetransfer. The

[14]It is recommended to use the filename of the library as a prefix for the function names (e.g. 'tunnel.pl' will use 'tunnel__' as prefix for each function).

function names have to be mapped, too. In contrast to the connection plugin, the filetransfer is assumed to be connection-less, hence there are no 'open' or 'close' functions. Additionally the connection to the respective GridServer must be opened prior to using any of the filetransfer functions

- **&ftp_init(\*%data*)**: Initialises the filetransfer method (lines 27 - 29). The hash contains the data of all items, which have been supplied upon installation and configuration (e.g. IP address, port, login, etc.).
- **&ftp_get(*$remote,$slot_id,$local)***: Copies a file $remote, which is stored in a remote temporary directory (identified by $slot_id but not known by its absolute name) to the absolute local path $local. In order to retrieve the absolute remote path, the corresponding server-command **{slot.workdir}**$_S$ is employed.
- **&ftp_put(*$local,$slot_id)***: Copies a file from the absolute location $local to the remote slot $slot_id. The retrieval of the absolute remote path is done in the same way as before.

The XML plugin-descriptor can be found in the appendix (Fig.A.13).

3.3.10 Processor plugin

The only purpose is to provide a convenient method for supplying a speed number for the numerous processor and to avoid, that the user has to provide it manually. Hence it does not require any libraries or commands by default and only needs a short XML descriptor. A sample thereof can be found in the appendix (Fig.A.12). It contains an absolute but arbitrary speed-number, which is obtained from several benchmarks (line 9) as its only mandatory item (apart from the ID). This supplied number does not have to be exact, because each program may use its own internal methods for benchmarking the resource similar to the **WIEN2k** plugin (see chapter 4).

3.3.11 W2GRID-packages

In order to share code with other users or to distribute bug fixes and new versions, it might be mostly sufficient to provide the commands, $\ulcorner JOBS \urcorner$ and libraries as an archive, download and expand them and perform some database-entries manually.
Yet the more convenient method provided by **W2GRID** supplies plugins, bug fixes and versions as a 'package', which does not only contain the source files, but also the instructions for their installation and removal. The instructions are read and executed by a **W2GRID** tool. If no input from the user is required, even very complex tasks can be performed without any interaction (see section 2.10).

3.3.11.1 Creating a package

First all commands, $\ulcorner$*JOBS*$\lrcorner$, libraries, tools, etc. have to be created and tested and have to be fully operable. The respective source code files, which shall be part of the package are supplied as a simple list, contained in an ASCII file (see the sample in the appendix: Fig.A.18). The installation instructions must be provided in an XML-coded form (Fig.A.16), which is separated into individual sections by pre-defined labels (e.g. ':etc' line 1). The available sections are:

- **:etc** Define the <header> and <footer> text for the begin and the end of the installation/removal (Fig.A.16 lines 1 - 3).
- **:envvars** Check environment variables. The respective value may be used in the script as $ENVenvvar-name (Fig.A.16 lines 4 - 14).
- **:variables** Interactively query the user. The entered results may be checked by regular expressions and the values used in the script as $VARvariable-name (Fig.A.16 lines 15 - 27).
- **:directories_install** Check/create a directory (Fig.A.16 lines 28 - 53).
- **:directories_uninstall** Remove a directory (Fig.A.17 lines 10 - 34).
- **:files_install** Check if a file exists.
- **:files_uninstall** Remove a file (Fig.A.17 lines 35 - 53).
- **:databases_install** Check/create a database.
- **:databases_uninstall** Remove a database.
- **:tables_install** Check/create a table.
- **:tables_uninstall** Remove a table.
- **:dbinserts** Make a table-insert (Fig.A.16 lines 54 - 69).
- **:dbdeletes** Delete entries from a table (Fig.A.17 lines 54 - 63).
- **:dbupdates** Update entries.

Each of the sections listed above can contain only specific XML-tags. The only floating one is the <SHELLCMD>, which allows to execute commands from an internal Bourne-Shell (sh) and have to be written in an appropriate syntax. They may be used to e.g. compile sources.

Example:
“<SHELLCMD begin = yes echo = ’compiling database’>cd $ENV{GRIDSRC}/SRC_wiensql/; make </SHELLCMD>” will recompile the database. This instruction may be placed into any section (see above). It is executed ***before*** *the other elements of this section, because of the setting ’begin=yes’. If it shall be run* ***afterwards*** *it needs ’end=yes’. Before executing the command, the message ’compiling database’ will be written to the screen.*

The three files (archive, install-instructions, removal-instructions) are turned into a package by the aid of the tool ***gridpackage_create.pl***. GridServer and GridClient packages have to be provided separately, because both daemons maintain different database accounts. If e.g. a GridServer package with the name WIEN2k_server† shall be created, the appropriate command is:

>***gridpackage_create.pl -signature SERVER WIEN2k_server***

A screenshot of the creation of such a package is provided in Fig.3.12. The text of this screenshot has been truncated to fit the given line-width of this document. The resulting package

```
> gridpackage_create.pl -signature SERVER WIEN2k_server

File, which contains the sources : wien_server.files
File, which contains the INSTALLATION-instructions: wien_server.install
File, which contains the REMOVAL-instructions: wien_server.uninstall
adding SRC_gridsrv/commands/wien/ ... done
adding SRC_gridsrv/jobs/wien/ ... done
adding libs_perl/wien/gridsrv ... done
adding libs_perl/wien/in1.pl ... done
adding libs_perl/wien/klist.pl ... done
adding libs_perl/wien/machines.pl ... done
adding libs_perl/wien/parameter.pl ... done
adding libs_perl/wien/struct.pl ... done
adding libs_perl/wien/wienfiles.pl ... done
adding libs_perl/wien/wien.pl ... done
adding libs_perl/wien/wienvar.pl ... done
adding bin/memory_lapw.pl ... done
adding bin/parameter_lapw.pl ... done
adding bin/calctime_lapw.pl ... done
FINISHED!
```

Figure 3.12: The creation of a **`W2GRID`** package, which will install the GridServer side sources of the **`WIEN2k`** plugin.

may afterwards be put on a web site and shared with other users.

3.3.11.2 Installation

In order to use this package it has to be copied into the directory $^\sharp$packages/. Because it was created with the option *’-signature SERVER’* only the GridServer will accept it. For installation, one may either use the menu-guided scripts or install it manually from the commandline. The latter is invoked with the command:

>***gridpackage_install.pl WIEN2k_server -authentication SERVER –install***

whereas the menu-guided approach starts with launching the master-script

>***cd $GRIDSRC/bin;gridsrc_install.csh***

and navigating through the menus to the appropriate one. Since it is a server-package, the user has to change to the server-menu [s][15] and and further to the package-installation [pi]. All available server plugins stored in the directory ♯packages/ are displayed in an enumerated list, unless they are already installed. The desired plugin has to be selected by typing its number (e.g. '1' for the WIEN plugin). The subsequent installation process will be exactly the same as the manual one. This procedure is explained by the aid of numerous screenshots in the usersguide. [97].

3.3.11.3 Removal

The removal, too may either be performed manually or by the aid of the same sub menu. The brute-force removal - simply deleting the files - is not recommended, since some leftovers in the database may confuse the daemon. The command for removing a package is:

>***gridpackage_install.pl WIEN2k_server -authentication SERVER –uninstall***

The menu-guided choice requires to select [pr] (package-removal) from the GridServer menu (instead of [pi] as before). All installed GridServer plugins are displayed again as an enumerated list. To remove the desired plugin, its number must be entered at the commandline. The following procedure will either employ the removal-instructions of the provided package or simply delete the list of the package-files if no uninstall-instructions are given.

[15] S + RETURN

Chapter 4

The WIEN2k application plugin

According to the Hohenberg-Kohn theorem, all properties of a crystalline solid can be obtained from the electron-density, which is calculated by **`WIEN2k`** in a self-consistent-field cycle (see Figure 1.1 in the introduction). A lot of observables can already be derived from the results

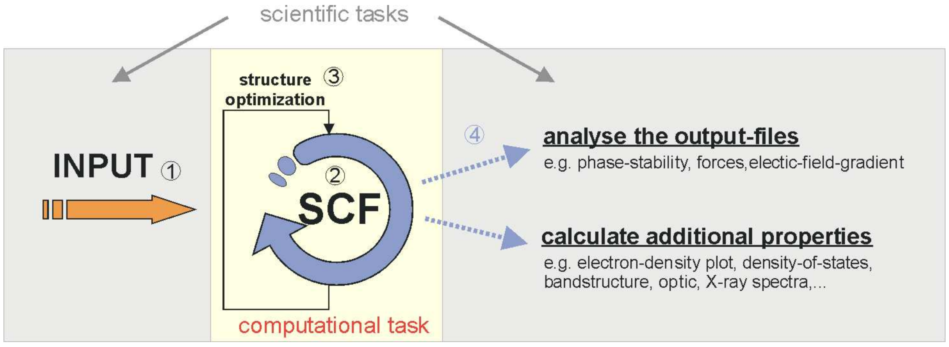

scientific tasks
INPUT ①
structure optimization ③
② SCF
computational task
④
analyse the output-files
e.g. phase-stability, forces,electic-field-gradient
calculate additional properties
e.g. electron-density plot, density-of-states, bandstructure, optic, X-ray spectra,...

Figure 4.1: Scientifi c computing with **`WIEN2k`**: Generation of input, SCF-cycle and analysis

of the SCF cycle, whereas other features of **`WIEN2k`** require to run additional programs afterwards. From a researcher's viewpoint, most chemical and physical knowledge is needed in the initial phase 1 $_{4.1}$, when the input is generated and afterwards 4 for interpreting the results or feeding them as input to other executables. The task in between, however, especially the actual execution of the SCF-calculation 2 can be demanding with respect to the computing time but require little interference from the user, so that this task can be coded with reasonable effort into proper scripts. Even a workflow like a structure optimisation 3 is from the computational point of view not more than a sequence of SCF-calculations, which can be automated too.

4.1 Purpose

For this reason the **`WIEN2k`** plugin is focused on the main task, namely to run SCF-calculations whereas the preceding tasks (i.e. the generation of the input) and the analysis of the results will remain unchanged and still needs to be done by the user in person. The plugin shall provide: A fully automatic submission of an SCF-calculation, which involves the evaluation of the given task (section 4.2.1), the selection of the most suitable machine (section 4.2.2), the filetransfer (section 4.2.3), the remote calculation-start (section 4.2.4), an intermediate file-transfer to update (section 4.2.5) the local output files and a proper cleanup after the calculation has finished (section 4.2.6). If an already submitted calculation does not yield the desired performance/results, there needs to be a reliable kill-method, which terminates all processes, that were invoked remotely. Finally the plugin has to offer a convenient interface to view status data of the individual **`WIEN2k`**-calculations. As a constraint for the selection of hosts, it is not allowed that parallelised SCF-calculations are run across several domains, instead they will either be submitted to a single managed cluster or several GridServers (slaves) of a virtual **`W2GRID`**-cluster.

4.2 Performing an SCF-calculation

The pseudo-workflow shown in Fig.4.2 illustrates the tasks, which are to be performed by the plugin. It shall be pointed out, that the procedure is exactly the same for the user as well as for **`W2GRID`**.

4.2.1 CASE-evaluation

Before any calculation can be started, the resource requirements have to be evaluated $\boxed{1}_{4.2}$ in terms of the estimated memory consumption (see section 4.4.1) and the respective runtime, which is obtained from an analytic performance model (see section 4.4.2). GridServers with insufficient total memory will be removed from the list of possible hosts already at the very beginning. The capabilities of the inbound network are important in the case of big lattices, which have only a few or even just a single k-point[1]. Since the k-point parallelism is not applicable in such a CASE, MPI must be employed. In order to yield a reasonable gain in performance, it requires a fast network (i.e Myrinet or Infiniband). The current version of the **`WIEN2k`** plugin does not use MPI-parallelism.

[1]The bigger a unit cell in real space the smaller it will be in reciprocal space. The integration is done in the reciprocal space, hence a smaller reciprocal lattice requires less k-points for integration.

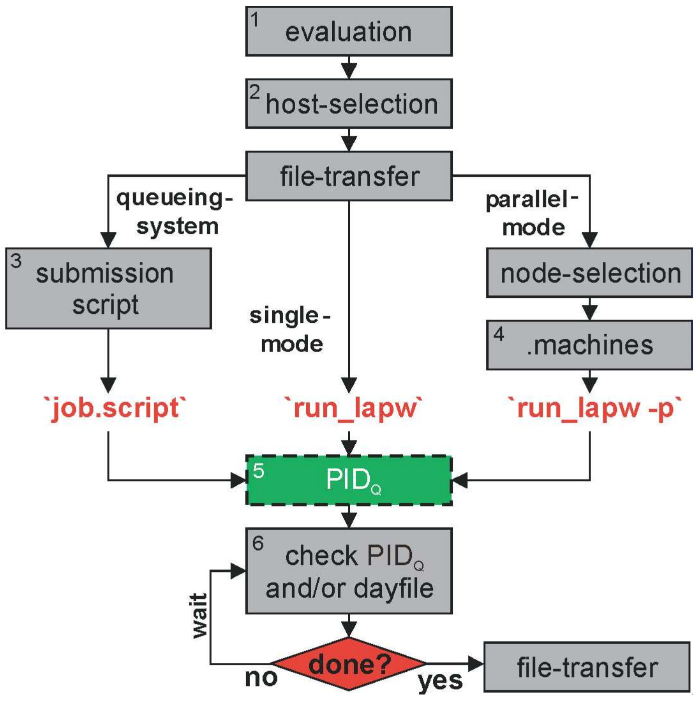


Figure 4.2: Administrative tasks of a **WIEN2k** calculation by the use of different types of job-submission schemes

4.2.2 Host selection

The selection - also referred to as 'match-making' - is based on the results of an evaluation, which should help to find the optimal resource for a given CASE. Contributions from the following aspects of the match-making are taken into account:

1. **Static data:** All informations, which describe the host and have been collected already in advance. Hosts incapable of running the respective CASE (e.g. due to insufficient memory or simply because **WIEN2k** is not installed) are omitted.

2. **Dynamic data:** This requires to contact the host and to request current status informations [98]. A host may have sufficient capacities (e.g. memory/nodes) and therefore passes the static evaluation, but can be "busy" at the moment. In this case it may either be worth waiting for its resources to become available within a reasonable time, or otherwise it is omitted, too.

3. **Preferences:** As a result from the dynamic analysis, each host offers a certain performance to run the given CASE. The match-making is completed by applying specific preferences (e.g. performance or total runtime).

4.2.2.1 Static

It is evident, that **`WIEN2k`** must be pre-installed and configured properly on each host, where a **`WIEN2k`**-job is supposed to be run. What seems to be obvious for a user, needs to be coded in the plugin. The GridClient keeps a long list of properties in its host-registry, one of these properties is a list of installed programs. The first restriction, which is applied to the selection of GridServers is the availability of **`WIEN2k`**. Such information about GridServers is automatically fetched by the job $\ulcorner host.check \lrcorner_C$, which keeps the registry up to date. This restriction is important for **`WIEN2k`**, because its installation is not trivial since it requires some expertise and cannot be done at runtime. Copying the numerous binaries before starting the calculation would be an other option which does, however, not make much sense for performance reasons.

The second important quantity is the memory: Each Fortran executable of an SCF-workflow consumes different amounts of physical memory, but ***lapw1*** consumes the most, hence it imposes the amount of memory, which has to be available for a good portion of the total runtime, otherwise the respective host will not be able to complete a single iteration within a reasonable time[2]. Consequently hosts with less or barely enough memory must be omitted. This evaluation is referred to as 'static', because the required information is available in advance.

4.2.2.2 Dynamic

The memory required by ***LAPW1*** has to be available at runtime and thus the host must be contacted to retrieve its current memory situation. At the same time, the load-situation is checked, too. This information -although obtained from different plugins- is formatted in such a way, that the **`WIEN2k`** plugin can easily retrieve the number of free nodes and their CPU usage. The load-situation influences the k-point distribution. On a managed cluster consisting

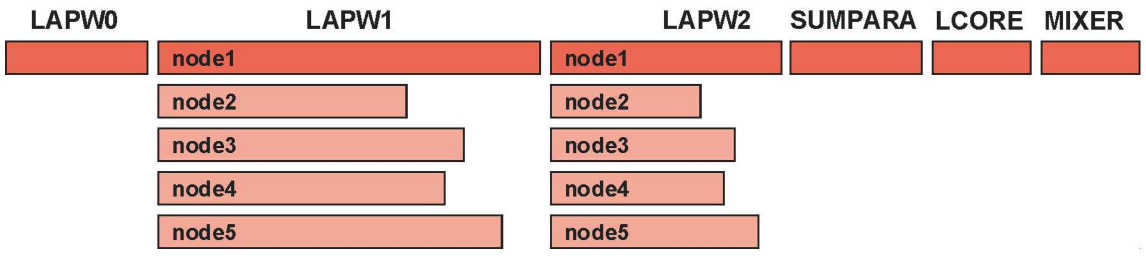


Figure 4.3: Contributions to the total runtime of a k-point parallel SCF-cycle (run_lapw)

of identical nodes, each node dedicated to the actual CASE will usually process the same

[2]The performance of a **`WIEN2k`** job is signifi cantly affected if swap memory is used.

number of k-points as long as the queuing system can guarantee that no other task is run on the very same CPU/core at the same time, which would drain performance from the node. On an unmanaged cluster or an array of individual desktop computers, the load situation may change frequently, therefore it cannot be taken for granted, that the CPU will only be exploited by the **`WIEN2k`** calculation. For this reason each machine is given an individual number of k-points, which is determined from the product of the current load, the power of the CPU and some factor. The resulting runtime will be different for each chunk, which is illustrated in Fig.4.3. It is desirable, that the runtimes of the chunks match as close as possible, otherwise the faster nodes will be idle until all remaining parallel processes of this step (e.g. LAPW1) are completed and the workflow can proceed to the next one (e.g. LAPW2). This waste of CPU cycles shall be avoided as much as possible, hence the **`WIEN2k`** plugin has to care for an optimal load distribution.

4.2.2.3 Preferences

In most cases it is the fastest host, which will be assigned to run the given CASE, but there are circumstances, why a host with inferior performance my be preferred though:

- The slower host/cluster can access the local CASE-directory directly by NFS. As a benefit, the filetransfer can be avoided. This choice yields an additional benefit, since the scientist will obtain the results in real time in the local directory, hence there is no latency between the remote generation and the local update of the data like in any other case.
- Sometimes it is a waste of resources to occupy the powerful machines with small or less important CASES, which are better served by a set of medium sized local desktops, while the powerful clusters can be kept for larger CASES.
- Submitting a CASE to the queuing system of a managed cluster is not always the best choice, even if the cluster offers an array of fairly powerful nodes. If this cluster is quite busy, the job will be queued for a long time. The corresponding probability that a job is granted the requested computing facilities within a certain time is difficult to estimate exactly. If this job can be run on a less powerful host but with a presumably shorter or even no queuing time it might be a better choice.
- If the memory is barely sufficient for the CASE, a second task submitted by another user during the runtime of the job will diminish its performance, hence a host with more RAM is preferred.

4.2.3 File transfer

If the selected host(s) cannot directly access the source code directory, the input files need to be transferred $\boxed{2}_{4.2}$. Usually it is not reasonable to copy the whole directory, especially if this CASE has already been run before and thus contains numerous (large) output files (e.g.CASE.vector) causing a significant and unwanted overhead. To avoid this, the files may either be selected manually (provided the corresponding expertise is given), or by the use of the shell-script ***migrate_lapw***, which supplies a list of all important input- and output files. This executable will be explained later in this chapter.

4.2.4 Starting the computation

The SCF-calculation can be run in two modes (see Fig.1.2), either single or parallel. The single mode is chosen if either a the executable ***lapw1*** takes less than a couple of minutes to complete (the adjustable threshold of the plugin is set to a default of 120 seconds) or, if only a single node is available. The parallel mode in contrast needs a properly formatted .machines†-file, which distributes the k-points. Depending on the kind of job-submission scheme, there are two different ways to generate this file. On an unmanaged cluster or on a set of desktops, the hostnames are known in advance, and the content of the .machines†-file $\boxed{4}_{4.2}$ will assign each host an individual number of k-points. It needs to be provided **before** the job is started. This is different to the situation on a managed cluster, where the nodes are **not known** in advance. In this case submitting a job requires to write a proper submission-script $\boxed{3}_{4.2}$, whose syntax has to conform to the requirements imposed by the installed queuing system. All corresponding instructions, such as how to create the .machines†-file from the supplied node-list[3] and how to run the application have to be coded in this submission-script.

4.2.5 Frequent checks

In either case (parallel or single mode) the invoked computation will be associated with a certain ***PID***$_Q$[4] $\boxed{5}_{4.2}$, which can be used to retrieve its status. With the proper architecture-dependent command it can be checked whether the calculation is queued, running or already finished[5]. In the latter case one needs to find out how the process has finished. For this purpose **`WIEN2k`** provides a logfile, which contains all important process-related informations. The final entries allow to tell whether the CASE has been accomplished successfully or terminated with errors.

[3]A list of the nodes, which are allocated for the job

[4]Process-ID. The lower-case 'q' refers to the fact, that this PID is only recognised by the queuing system and is distinct from the PID of the operating system.

[5]Either successfully or with an error

4.2.6 Cleanup

Usually it is necessary to copy the output files from the remote directory back to the local one, unless they are identical. As mentioned before, the executable ***migrate_lapw*** can be used to retrieve the names of the files in question. In most cases, the remote files can be discarded afterwards.

4.3 Interfaces

As explained in the introduction, **`W2GRID`** is supposed to be a non-invasive infrastructure for distributed computing, hence the application plugin is restricted to the same mechanisms for interacting with **`WIEN2k`**, which are also available to the user, however, a few non-critical changes and extensions were required, mainly to enhance the data-acquisition. These changes are minimal and can of course also be used without **`W2GRID`**. Basically they serve for a better maintainability of the plugin than any alternate solution. None of them will affect the regular use of **`WIEN2k`**, thus the application can still be run as before without **`W2GRID`**. The interfaces are explained in the following subsections. Any required changes or extensions are marked explicitly.

4.3.1 migrate_lapw

This is a C-Shell script, which provides a list of CASE-related **input-** and **output-**files, if invoked with the additional flag:

$>$***migrate_lapw -show***

Originally this script was intended to transfer the input of a CASE to a remote location and copy the results back to the local workstation. This functionality could not directly be employed, but since the list of files can easily be extracted, the script was modified in order to recognise this flag and return the list. A sample output is shown in Fig.4.4. The files listed as 'file_end_append:' indicate a special type of output files, which are not re-written after each iteration. Instead they grow continously during the runtime of the calculation. In order to avoid the overhead of transferring these files with each update, the plugin is able to request just a chunk of its content, namely exactly that piece which is missing locally from from the remote GridServer and append it to the existing stub in the local directory.

```
file_start:
tic.struct tic.clmsum tic.clmup tic.clmdn tic.klist tic.kgen tic.dmatup \
tic.dmatdn tic.vorbup tic.vorbdn tic.scf tic.broyd1 tic.broyd2 .machines *.in*

file_end_append:
tic.dayfile tic.scf

file_end_optional:
tic.vector* tic.energy* tic.help*

file_end_default:
tic.struct tic.in1* tic.in2* tic.clmsum tic.clmup tic.clmdn tic.clmval* tic.dmat* \
tic.clmcor* tic.vorb* tic.broyd* tic.output0* tic.output1* tic.outputso* *.error \
tic.output2* tic.outputdm* tic.outputorb* tic.outputc* tic.outputm
```

Figure 4.4: Sample output of the C-Shell script ***migrate_lapw***

4.3.2 testcomplex_lapw

This is a C-Shell script, which analyses, whether a CASE leads to complex or real matrices, simply by checking the existence of the file CASE.in1c†. Being complex means that the CASE has an increased memory requirement and also affects the runtime. This is the only script that had to be newly generated for **W2GRID**.

4.3.3 run_lapw, runsp_lapw, runafm_lapw, ...

These are several C-Shell scripts, which represent a workflow similar to Fig.1.2. In addition to the few Fortran executables illustrated so far, there are others like ***LAPWDM*** or ***LAPWSO***. Since **W2GRID** does not interfere with the internal mechanisms of the shell-scripts, their individual tasks and contents do not need to be discussed here as they are also concealed from the plugin. Details about the SCF-cycle, its individual components and its input are provided in the **WIEN2k** usersguide [9]. To start a calculation, the respective command

>***run_lapw [OPTIONS] [FLAGS]***

is invoked either directly on the commandline 4 $_{4.2}$ or from within the submission-script 3 . Parallel (k-point and/or MPI) processes require the flag '-p' and the presence of the .machines† file. Other flags and options are available from the built-in help.

4.3.4 .machines

This is the data file for parallelisation, which contains the list of all hosts/nodes to be used and the number of k-points, which are assigned to them. The content of the sample file in Fig.4.5 assumes a total number of 10 k-points[6]. In the given example **hostA** will process two times

[6]Basically the numbers before the colons are just weights, and the k-points are distributed accordingly as a share. The given example will agree also with any number of k-points, which is a whole numbered multiple of 10.

```
2:hostA
4:hostB
1:hostC
3:hostD
```

Figure 4.5: Sample content of a .machines† fi le

as many k-points than **hostC**, which is either due to the CPU being two times as powerful as the other one or due to the current load situation, which allows **hostA** to exploit 100% of the CPU capacity, whereas **hostC** gets only 50%. Additional parameters (such as *extrafine*, *granularity* or *residue*) can be used to tune the load balance. These parameters do not change the number of k-points assigned to the individual hosts and thus do not need to be discussed here. For further details the reader is referred to the **WIEN2k** usersguide [9].

4.3.5 .stop

Each workflow-script (e.g. ***run_lapw***) will check for the presence of this command-file at the end of an iteration before starting the next one. If the file exists in the CASE-directory, the workflow is stopped. A corresponding entry in the CASE.dayfile† will indicate that the calculation has been cancelled by the user. The reason for terminating the (mostly numerous) processes invoked by **WIEN2k** by this command-file is, that the result of the current iteration will not be overwritten and thus the user does not risk corrupted output files, which might result from a brute-force termination. Yet the process is not stopped immediately but with a delay of at most a single iteration, which can be considered to be a disadvantage of this scheme.

4.3.6 lapw1

The Fortran executable, which performs the time-consuming set-up and diagonalisation of the Hamiltonian matrix. The size of this matrix is a key quantity for the performance model and the estimation of the memory requirement. This number is calculated at the beginning of ***lapw1*** but originally was not displayed, hence the executable had to be modified in order to obtain this value. The new version of ***lapw1*** will accept an additional input-parameter, which has the effect to only write the desired MATRIX-size to a file (CASE.nmat_only†) but to quit afterwards without commencing the set-up or the diagonalisation. A user, who wants to apply **W2GRID** and the **WIEN2k** application plugin either has to install the new version of **WIEN2k** or just replace ***lapw1*** with the new version and recompile it. Instructions for how to recompile individual executables of are provided in the usersguide **WIEN2k** [9].

4.3.7 x_lapw

All **WIEN2k** binaries are enveloped by a powerful C-Shell script, which manages the complex input. ***x_lapw*** (aliased to ***x***) takes the name of the executable (e.g. ***lapw1***) and several input-parameters as arguments to create a single input file (e.g. lapw1.def), which contains all important filenames and Fortran filehandler-numbers that are used internally by the executable. Afterwards the program is executed [9]. To retrieve the MATRIX size, the following command must be invoked:

>***x lapw1 -nmat_only***

The additional flag '-nmat_only' did not exist in the original implementation of **WIEN2k** and had to be included into the code.

4.3.8 CASE.nmat_only

This file contains the MATRIX-size that was determined in a short run of ***LAPW1*** as described above.

4.3.9 input files

Certain files (e.g. CASE.in1†[7]) are required to retrieve some calculation parameters. The filenames and their content are summarised in Table 4.1.

4.3.10 .parameter

```
matrix:1867
k:12
atoms:4
complex:0
lm:39
```

Figure 4.6: Sample content of a .parameter† file

W2GRID retrieves all parameters that are required for estimating of the computational effort (memory and runtime) and writes them to the file .parameter†. Its purpose is to allow a convenient retrieval of the data without the need to compute and extract the data again. By comparing the modification dates of this file and the respective input files, **W2GRID** can tell whether the .parameter† is still valid or needs to be updated. A sample content is shown in Fig.4.6 and shows (in this order) the matrix size, the number of k-points, the number of non

[7]A complex CASE will use the filename CASE.in1c instead

equivalent atoms, the type of the matrices (complex or real) and the number of lm components in the spherical harmonics expansion.

4.3.11 .w2grid_lock

This command-file is created in the CASE-directory by **W2GRID** after the **WIEN2k** plugin has started to process the files. It is removed again after the calculation has finished. This file serves to signal another programs or scripts, that the directory is in use. In the scope of a larger workflow (see section 4.5.6) it is needed to pause a loop of sequential calculations.

4.4 Resource evaluation

4.4.1 Memory requirement

An estimation of the required memory allows to pre-select possible hosts from the list of Grid-Server characterised with 'static' parameters. Since LAPW1 is the most memory consuming workflow element, only its maximum size has to be calculated, whereas the other programs can be omitted due to the fact that the executables are run sequentially. The memory requirement of LAPW1 can be obtained with equation 4.1, whose symbols are explained in Table 4.1.

$$size = M^2 \cdot 8 \cdot c \cdot (bytes) \qquad c = \begin{cases} 1 & \text{real, (inversion)} \\ 2 & \text{complex, (no inversion),} \end{cases} \tag{4.1}$$

When the plugin scans the registry for an appropriate host the $size$ must not exceed 80% of the static total memory of a machine (or a single node of a cluster), otherwise this resource must be omitted.

4.4.2 Analytic performance model

The computational effort is more difficult to obtain, since numerical methods are only poorly accessible to analytic approaches. The intrinsic constraints are listed below.

- The time for a single iteration of an SCF calculation can more easily be estimated than the number of cycles, which depends on the desired accuracy as well as on the accuracy already achieved in previous runs[8].
- A single iteration employs numeric methods and thus cannot be timed precisely. Furthermore each CASE is unique and thus a general approach of a given performance model

[8]An already converged calculation, which is run again with higher accuracy, usually needs fewer iterations

can never fully cover all different CASES.

- The total runtime is a sum of contributions from several independent Fortran executables (see Fig.1.2), of which 'LAPW1' is the dominant one, so that the runtime of the others can be estimated as a fraction thereof. Therefore the performance model will calculate the runtime of 'LAPW1' according to eq. 4.5 and account for the additional contributions from the other executables by multiplying this result with a constant factor (see below).

- The runtime of a CASE depends in the first place on the machine-performance with respect to the numerous floating-point- and I/O operations, whose numerous architecture dependent contributions can practically not be taken into account. Instead the overall performance is abstracted by a single scalar constant: A machine-speed f supplied once at the installation of **`W2GRID`**. Note that this number can be retrieved by every plugin, independent from the application it is supposed to serve. Since this is only a poor approximation of many different contributions, an additional application-dependent tuning factor g is necessary. It is up to each plugin, whether this factor is used or not. In the case of **`WIEN2k`** it results from an automated self-benchmarking, where every completed SCF calculation refines the value. This factor is supposed to correct the insufficient machine speed f and account also for the specific compilation of the **`WIEN2k`**-code on a given host, which can cause a gain or loss in performance (e.g. 32bit vs. 64bit executables).

A crude performance model is sufficient, since the main purpose is to estimate the order of magnitude, whether a job runs for minutes, hours or days. This shall help to make a proper decision on the required computing resources rather than time the process exactly.
In order to estimate the runtime of LAPW1 it is necessary to inspect the different contributions. The generalised eigenvalue problem is set up and solved, which comes with three major contributions to the runtime: Setting up the Hamiltonian matrix (**HAMILT**; equation 4.2), adding the non spherical potential terms to this matrix (**HNS**; equation 4.3) and diagonalizing it (**DIAG**; equation 4.4). When a CASE is run, the executable writes the time consumptions of these parts to the file 'CASE.output1', whose content can be used to test and improve the model.

symbol	*purpose*	*origin*
n_{at}	number of atoms	from 'CASE.struct'
M	matrix-size	**x *lapw1 -nmat_only***
n_{LM}	angular momentum contributions	from 'CASE.vns'
α, β and γ	fitting parameters	fit from sample cases
k	number of k-points	from 'CASE.klist'
f	nominal machine-speed	**`W2GRID`** (installation)
g	correction factor for the speed	**`WIEN2k`** plugin (adaptive)
c	constant: 1 (real) or 3.2 (complex)	from the symmetry
t_{LAPW1}	total runtime of LAPW1	equation 4.4

Table 4.1: Symbols used for equations 4.2-4.5

$$\mathrm{t}_{HAMILT} = \alpha \cdot n_{at} \cdot M^2 \cdot c \tag{4.2}$$

$$\mathrm{t}_{HNS} = \beta \cdot n_{at} \cdot \frac{25 + \frac{n_{LM}}{n_{at}}}{25} \cdot M^2 \cdot c \tag{4.3}$$

$$\mathrm{t}_{DIAG} = \gamma \cdot M^3 \cdot c \tag{4.4}$$

$$\mathrm{t}_{LAPW1} = f \cdot g \cdot k \cdot (t_{HAMILT} + t_{HNS} + t_{DIAG}) \tag{4.5}$$

To obtain the total runtime of an SCF calculation, the result of equation 4.5 will be increased by $+40\%$ to account for other executables and multiplied by 20, an average number of iterations to reach self-consistency.

4.4.2.1 Test cases

The properties of interest, which shall be evaluated by test cases are:

- The machine-speed f (initially the adjustable tuning factor g is set to '1') of equation 4.5.
- The value of the complex-parameter c in the equations 4.2-4.4, which accounts for the fact that a complex takes longer than a real arithmetic.
- The three constants α, β, γ in the equations 4.2-4.4.

Since the compiler technology, the third-party libraries and hence the performance of the code improved over the years it is hardly possible to reconstruct the detailed conditions of CASES, which have been run in the past. Therefore 73 material science studies were selected and run again in single mode on four different hosts. The runtime statistics stored in the file CASE.output1† were used to fit the performance parameters. To obtain the nominal speeds (f), the slowest host is set to the arbitrary value '500' and the individual runtimes of all CASES are compared to the runtime of the slowest. The median of the resulting factor is used as machine-speed. The complex-parameter c is determined on the slowest host by a least-square fit of equation 4.2. Finally the three constants α, β, γ are optimised by a least-square fit and averaged over the hosts. The correction factor was ignored ($g = 1$).

$fitting-parameter$	$result$
α	$2.6 \cdot 10^{-5}$
β	$7.7 \cdot 10^{-6}$
γ	$8.6 \cdot 10^{-7}$

Table 4.2: Results for the fi tting parameter

The given fitting parameters yield an average accuracy of $^{+}/_{-}$16% for the inspected 73 cases. With these parameters one can, for example, estimate (according to eq. 4.4) that the diagonalisation of a complex CASE with a matrix size of $M = 10000$ will take roughly 5300 seconds (or 1.5 hours) on a Pentium IV (2.66 GHz) with a nominal speed of $f = 520$ to perform the diagonalisation.

4.5 Design of the application plugin

According to section 3.3.5 the plugin only consists of **{COMMANDS}** (section 2.6.2), $\ulcorner\llcorner JOBS \urcorner\lrcorner$ (section 2.6.3) and certain (optional) Perl-tools. None of its components need to be compiled and thus the scripts can easily be copied into the respective $GRIDSRC (sub) directories of the target hosts. Since the concept of **`W2GRID`** is RPC based, there are both, server-side as well as client-side stubs, hence the plugin actually consists of two parts which need to be installed on the GridServer and the GridClient, respectively. The essential command **{wien.exec}**$_C$, which submits and controls a calculation, involves most of the implemented stubs. It is illustrated in Fig.4.9. For the implementation of the plugin, no other changes and extensions than those already mentioned in section 4.3 are made to the code. Especially no Grid-routines or libraries are included into the **`WIEN2k`**-code, according to the definition of a lightweight infrastructure described in the introduction. It is up to the developer of a plugin to decide, which changes (if any) have to be made in the application, since **`W2GRID`** does not make any restrictions. In the case of the **`WIEN2k`** plugin it was decided, that all relevant parts, which may be subject to future modifications (such as a change of the list of input- and output files) remain in the responsibility of the **`WIEN2k`** code and are not implemented directly in the plugin.

4.5.1 Installation

4.5.1.1 WIEN2k

WIEN2k must be available for both daemons, the GridServer as well as the GridClient. While the GridServer runs the calculation, the GridClient must in the beginning evaluate a CASE and extract the required parameters. For example the retrieval of the MATRIX-size requires the execution of ***lapw1***[9] by the GridClient, which additionally has to obtain the list of input- and output files by executing ***migrate_lapw***.

- **Install WIEN2k from Scratch:** If **WIEN2k** does not exist on the respective host, the installation must be performed from scratch. Instructions how to do this are given in the **WIEN2k** usersguide [9].
- **Update:** In order to use the plugin, the recent **WIEN2k**-sources need to be obtained from the vendor[10] and copied into the proper directory. A subsequent compilation of ***lapw1*** (♯$WIENROOT/SRC_lapw1/) is mandatory.

4.5.1.2 Application plugin

The application plugin comes as a package for each daemon (see section 2.10) and will need to be copied into the directory ♯packages/. It is expanded and configured by the aid of the package-installer (section 3.3.11), which runs the provided installation routines and performs the following tasks:

- Check for the environment variable $WIENROOT and cancel the installation, if it is not set.
- Check, if the $WIENROOT-directory is still valid.
- Check the **W2GRID** directory tree and create the subdirectory ♯wien/ in the proper command- and job-directory of the respective daemon (see command-groups in section 2.6.2.2).
- Create the subdirectories ♯libs_perl/wien/gridsrv/ and ♯libs_perl/wien/gridclient/
- Copy the Perl-tools to ♯bin/
- Copy the **{COMMANDS}**, ⌈*JOBS*⌉ and libraries into the subdirectories, which have previously been created for this purpose.

[9] The GridClient could also delegate the MATRIX-size calculation to any of the registered GridServers, if **WIEN2k** is not installed locally. Such an approach might be considered in the future.

[10] http://www.wien2k.at

4.5.2 GridClient

The selection of the optimal host is mainly governed by the total runtime, but there are also contributions from node- and memory utilisation as well as the estimated queuing time[11]. In order to find the best host, the GridClient will first check static data such as the total memory and exclude those hosts, which are not capable to run the job anyway. Then it will request a proposal for the runtime and other job-submission related parameters as well as the intended load-distribution (k-point parallelism). For debugging purposes it is required, that the user may check these 'runtime-proposals' made by the GridServers without actually submitting the calculation. In order to perform all the listed tasks, the GridClient plugin offers the following commands:

- **{wien.mkmachines}**$_C$ allows to preview the proposals (number of nodes, total runtime), which the GridClient uses to choose the optimal host from among all capable ones, see sample output in Fig.4.7.

- **{wien.exec}**$_C$ submits a CASE to the optimal host (The workflow is illustrated on the left-hand side of Fig.4.9). It will return a unique identifier (**(slot_id)**$_C$), which can be used as argument for other commands (e.g. **{wien.list}**$_C$). The command will write an empty file '.w2gridlock' to the CASE-directory. This file will not be removed unless the calculation is done, hence it can be used to pause a larger workflow (see section 4.5.6). After having reserved the required resources on the GridServer (section 2.6.5.1) and the GridClient, the command quits and returns the client-side ID (**(slot_id)**$_C$) to the user. The time-consuming tasks (filetransfer and submission of the calculation) will be done in the background by a $\ulcorner JOB \lrcorner$. $\ulcorner wien.exec \lrcorner_C$ is invoked by the command **{wien.exec}**$_C$ for the first time just before the command quits and writes its output to the logfile (CASE.log†), whose content is XML encoded and may be read by the aid of the utility ***logview.pl***.

  >logview.pl CASE.log

  . After having started the remote calculation, the same $\ulcorner JOB \lrcorner$ is invoked in regular intervals in order to monitor the remote process and transfer the files if necessary. If the remote calculation is finished, the $\ulcorner JOB \lrcorner$ will clean up, set the local slot state accordingly either to 'FINISHED' or 'ERROR' and de-register itself.

- **{wien.list}**$_C$: All calculations are stored in the table 'slots.client', which is a persistent data container that preserves data beyond the end of the $\ulcorner JOB \lrcorner$ (see Fig.2.20). This command retrieves slot-informations from this table, which are labelled to belong to the **WIEN2k**

[11]Waiting half an hour to run a minute-job on a busy cluster is not acceptable, but for a big job, which takes several days the queuing-time will not matter.

plugin (column **(program)** of Table 2.4). If submitted without argument it will display all **`WIEN2k`**-calculations (running, erroneous and finished). To restrict the output to a single entry, the **(slot_id)**$_C$ as returned by the command **{wien.exec}**$_C$ may be supplied as an argument (see sample output in Fig.4.8).

- **{wien.check}**$_C$ While $\ulcorner$*wien.exec*$\lrcorner_C$ performs the check and filetransfer operations in regular intervals, this command does the same instantly on user-demand, which might be of interest for steering. It needs the **(slot_id)**$_C$.

- **{wien.kill}**$_C$ If supplied the **(slot_id)**$_C$ of a running calculation, the latter will be stopped immediately by triggering a safe kill-method on the remote host.

- **{wien.clean}**$_C$ The table 'slots.client' is a persistent data container for any kind of data relevant for a certain calculation. If entries shall be deleted, they must be deleted on purpose either by the use of the command **{slot.kill}**$_C$ or more conveniently with the respective plugin-command **{wien.clean}**$_C$. Without additional parameters it will delete all finished and erroneous calculations and leave only the running ones. if given a **(slot_id)**$_C$, only the respective entry will be removed.

```
>wien.mkmachines
#command took 8 seconds to complete
**********************************************
HOST           athena
**********************************************
RATING         1124 units
.machines      (one entry means single-mode)
               17:eos
               16:eos
               14:ne
               12:iris
               11:susi
COPY           NO
FREE SLOTS     30
MEMORY         1%
EFFICIENCY     45% (node efficiency)
ITERATIONS     20
START          estimated in 0 s (0+00:00:00)
SINGLE:        running on eos
               1 k-point     5 seconds (00:00:05)
               1 iteration   357 seconds (00:05:57)
               1 scf cycle   7140 seconds (01:59:00)
PARALLEL:      running on 5 machines
               1 k-point     1 seconds (00:00:01)
               1 iteration   87 seconds (00:01:27)
               1 scf cycle   1740 seconds (00:29:00)

COMMENT        Access to the local CASE-DIRECTORY is given.
               This improves the rating.
```

Figure 4.7: A sample output of the command **{wien.mkmachines}**$_C$

```
>wien.list 31
#command took 0 seconds to complete
1 [0 running, 1 finished, 0 queued, 0 error]

[31] tio2 FINISHED
    SERVER   :gescher
    LOCAL DIR:/athena/hschweifer/lapw/tio2
    SUBMITTED:19:52:24 24-08-2006
    STARTED  :19:52:40 24-08-2006
    FINISHED :20:15:52 24-08-2006
    END      :finished properly
```

Figure 4.8: A sample output of the command **{wien.list}**$_C$

4.5.3 GridServer

The purpose of its stubs, which are shown on the right-hand side of Fig.4.9 is to complement the workflow of the GridClient. The input as well as the output is XML-formatted. All commands will need the local resource identifier[12] (**(slot_id)**$_S$) to be supplied by the GridClient. Invisible and inaccessible to the user, the **GridServer** employs heuristic methods for improving the runtime-prediction (by adjusting g in eq. 4.5 using the obtained runtimes of each finished calculation). In order to run executables in parallel, a.machines†-file has to be created. An adaptive load-distribution mechanism allows to react to changing load-situations and to optimise the resource utilisation by adapting the contents of this file. MPI parallel processes are not implemented at the moment, but threading can already be employed.

- **{wien.exec}**$_S$ starts a calculation[13]. The command just prepares everything for the **`WIEN2k`**-job and delegates the frequent checks to the job $\ulcorner wien.exec \urcorner_S$. This interplay of command and $\ulcorner JOB \urcorner$ is comparable to the respective pair **{wien.exec}**$_C$ and $\ulcorner wien.exec \urcorner_C$. The **(job_id)**$_S$ is returned to the client.
- **{wien.kill}**$_S$ will properly terminate all invoked WIEN2k tasks immediately.
- **{wien.benchmark}**$_S$ benchmarks the local binaries and adjusts g (see table 4.1). It extracts the actual runtime of ***lapw1*** from CASE.dayfile† and compares it with the one calculated by the aid of the performance model (see equation 4.5) yielding a new factor g_{new}. The factor g_{old} stored in the table 'programs.server' will be updated according to equation 4.6, where a maximum change of 20% is allowed for each completed SCF-calculation. The contribution must be related to the calculated total-runtime t_{SCF} (seconds) since long-running CASES better conform to the runtime model than shorter ones ($T_{SCF} < 60s$). 5 minutes (300 seconds) are considered to be a sufficiently long runtime to

[12]which must have been obtained previously from executing the command **{slot.reserve}**$_S$

[13]An additional flag (-machines) will cause it to return just a .machines†-file proposal for **{wien.mkmachines}**$_C$ instead of performing the calculation.

diminish contributions I/O effects.

$$g = 0.8 \cdot g_{old} + 0.2 \cdot \frac{g_{old} * 300 + g_{new} * t_{SCF}}{300 + t_{SCF}} \tag{4.6}$$

4.5.4 Interplay of the command-stubs

Many individual commands on both daemons contribute to certain longer workflows. An example is shown for the workflow, invoked by **{wien.exec}**$_C$, which is illustrated in Fig.4.9. It contains also other commands than the already explained **`WIEN2k`**-specific ones. These are explained in the sections 2.6.5.1 and 2.6.3 as well as in the usersguide [97].

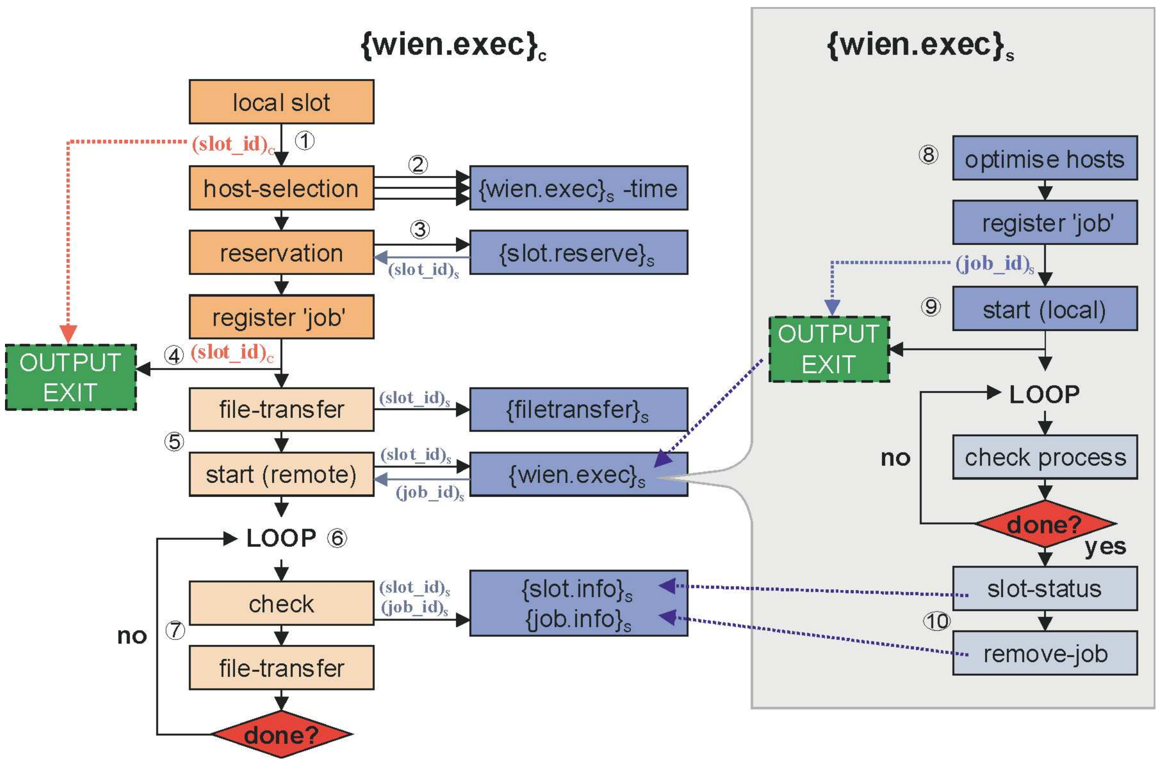


Figure 4.9: Workflow of **{wien.exec}**$_C$: Interplay of GridServer and GridClient commands/jobs

1 Each calculation is stored in a table and identified by a unique number, the **(slot_id)**$_C$. The entry contains a reference to the directory of the source code files and data about the job-name and calculation parameters. Later the data are appended with the **(slot_id)**$_S$ of the GridServer and the current status informations. The **(slot_id)**$_C$ is a mandatory/optional argument for **{wien.list}**$_C$, **{wien.kill}**$_C$, **{wien.check}**$_C$ and **{wien.clean}**$_C$.

2 A matchmaking process of the registered GridServers and the estimated memory requirement leaves only those hosts, which are capable of running the job. The GridClient

will then extract all CASE-related calculation parameters k, M, c, n_{at}, n_{LM} (see Table 4.1) and query all capable hosts in parallel[14]. The GridServers employ the analytic performance model and generate a proposal for a k-point distribution and the estimated total-runtime with respect to their given load-situation. The GridClient ranks the individual proposals and takes the top-ranked GridServer.

3 In order to identify a remote calculation, a resource has to be reserved. This resource, which is referred to as 'slot' is a database-entry, which contains some temporary data and a pointer to a temporary directory. This ID, the remote **(slot_id)**$_S$ is only used by the GridClient.

4 When all foreground tasks are done, the command will return the **(slot_id)**$_C$ as an output to the user. The workflow is continued in the background by $\ulcorner\!\llcorner wien.exec \urcorner\!\lrcorner_C$, whose output is written to the slot-registry and can be retrieved by the aid of **{wien.list}**$_C$.

5 In the background, the actual input files are obtained from the executable 'migrate_lapw' and copied to the remote resource ('**(slot_id)**$_S$'), then the calculation is started and the GridClient receives the corresponding **(job_id)**$_S$ of the GridServer background process.

6 The following loop will monitor the ongoing calculation and care for the file synchronisation. The interval between each run of this $\ulcorner\!\llcorner JOB \urcorner\!\lrcorner$ depends on the configuration of the GridClient and is usually in the range from 60-120 seconds.

7 The GridClient does not receive the remote ***PID***$_Q$, since it would not know how to retrieve the actual status. Hence this task will be left to the GridServer ($\ulcorner\!\llcorner wien.exec \urcorner\!\lrcorner_S$), which is supposed to monitor the process and to update the corresponding slot accordingly 10. This status can be IDLE, RUNNING, FINISHED or ERROR. The output files transferred during or at the end of the calculation depend on the status. As long as the calculation is RUNNING, intermediate results will be fetched, whereas the complete output will be copied only at the end (FINISHED or ERROR). During the runtime, updated input files are copied in the other direction (from the GridClient to the GridServer), which allows to steer the calculation. This can also be applied to transfer the file .stop† or to intentionally change the settings in the .machines† file.

8 The optimisation of the list of hosts depends on the performance model. The GridServer will adaptively improve its correction factor (g in Table 4.1) to yield a better prediction. The list is obtained by fetching the list of free resources (nodes) and iteratively improving

[14]**{wien.exec}**$_S$ -machines

the k-point distribution until the runtime is optimal with respect to the given constraints[15].

9 The GridServer uses its 'execution plugin' (see section 3.3.7) for submitting the calculation ('exec__submit') and receives a local ***PID***$_Q$, which is stored as a string and will only be used locally. The function '&exec__existspid()' comes to play in the retrieval of the calculation state.

10 The status of the remote-slot represents the state of the calculation (see above). The status is set by the job $\ulcorner wien.exec \urcorner_S$, but it may happen, that this $\ulcorner JOB \urcorner$ fails to update the slot properly due to some problems. Therefore the status of a slot only makes sense if combined with the state of the $\ulcorner JOB \urcorner$ that must exist as long as the slot is RUNNING but has to be gone, once the slot-status is either FINISHED or ERROR. Other combinations will be interpreted as FATAL and trigger the error handling. The treatment of such cases is up to the developer. The **WIEN2k** plugin repeats this check two times and then finally quits with an error.

4.5.5 Tuning the parallelisation

In most cases the expertise of a user cannot be replaced by an automatic system, but the application plugin provides several parameters to influence the proposals for the .machines† file. The parameters can be supplied as arguments[16] to the commands **{wien.exec}**$_C$ and **{wien.mkmachines}**$_C$ and are forwarded to **{wien.exec}**$_S$.

- **mintime:** If the workload is shared on several nodes, each chunk will take a certain and maybe different time to run, (depending on the speed of the node and/or the number of k-points assigned to it). Theoretically (assuming, that all nodes are free and that there are enough k-points) **W2GRID** could spread the task on all available nodes, but often is not desirable to have a single CASE occupy all available resources (in a multi-user environment this is mostly the case), then the number of involved hosts can be limited. This can be done by setting a minimum runtime. The optimisation of the proposal always starts with all available hosts. If the runtime of a chunk is less than the minimum and cannot be increased by shifting a k-point from a slower node to the faster one without causing the other runtime to drop below this limit, the slowest host[17] is abandoned and its k-points are equally distributed on the others. The default for this parameter is set to 125 seconds.

[15]The minimum runtime of a single chunk of k-points is limited, as well as the total number of hosts and the maximum CPU load.

[16]e.g. '-runtime 20' will set the lower bound of the runtime of a single chunk to 20 seconds.

[17]The slowest host is determined by comparing the runtimes of all hosts for a single k-point. The host, which would take longest offers the least performance.

- **cpulimit:** **W2GRID** needs to distinguish on the one hand a ’busy’ CPU from a ’free’ one and on the other hand must properly handle multicore or multi-processor systems. **W2GRID** is able to forecast the estimated CPU usage (in %) a process would be allowed to exploit if started on a certain host[18]. If three processes run on a single-core CPU, each will use 33% of the total CPU power. If it is a dual-core CPU, each process gets 66%. The cpulimit is the threshold for this forecast. If the estimated CPU usage drops below this threshold value, the machine will not be used [19]. This prevents that a dozen of **WIEN2k** jobs are submitted at the same time to the same node/CPU. The default is set to 66% and allows that two tasks may be run in parallel on a dual-core but not on a single-core CPU.

- **maxhosts:** The purpose of this parameter is related to ’mintime’, yet this one allows to restrict the total number of hosts employed for big jobs, where the runtime of a single chunk does not undercut ’mintime’. If not constrained by the ’cpulimit’, the plugin will therefore take all free nodes for the CASE. If this argument is provided (e.g. ’-maxhosts 6’) the host-list, which is sorted by the runtimes starting with the fastest ones, will first be cleared of the slowest excess hosts and limited to ’maxhosts’ machines. The remaining list will be optimised as usual.

4.5.6 Development of larger workflows

’Phonon-calculations’ or ’structure optimisations’ [9, 12] will be composed of individual SCF-calculations being run either sequentially or in parallel. Since the commandline interface supports a batch mode (-S flag) and because the application plugin can invoke any of the existing **WIEN2k** workflows (e.g. run_lapw, runsp_lapw, ...), the client-side commands can easily be integrated in C-Shell scripts. A sample code for a structure optimisation is provided in Fig.4.10. The use of .w2grid_lock[†] is demonstrated in lines 6 - 8. As long as the file exists,

```
1: #!/bin/csh -f
2: foreach i (FePt3_vol_-3.0 FePt3_vol_0.0 FePt3_vol_3.0)
3:     cp  $i.struct FePt3.struct
4:     set RPC = "wien.exec -program runsp_lapw -parameter ’-cc 0.001’"
5:     gridclient.pl -S "$RPC"
6:     while (-e .w2gridlock)
7:          sleep 60
8:     end
9: end
```

Figure 4.10: Code sample of a volume-optimisation with **W2GRID**

[18]This estimation is always calculated per host/node, consequently one dual-core CPU will be treated in the same way as two single-core CPUs.

[19]or more precisely: no further process will be added to this host/node since a single host can receive more than a single chunk (provided it has more than one CPU or core).

the workflow is paused and no additional calculation can be submitted to **W2GRID**. If the recent SCF-calculation is done, the file is removed and the workflow of the script proceeds.

4.5.7 Tools

The plugin also comes with certain tools, which are implemented to debug the respective libraries. They will be copied to ***$GRIDSRC/bin/***.

- ***parameter_lapw.pl*** extracts the calculation parameters from the input files and runs ***x lapw1 -nmat_only*** if required. It creates the data file .parameter†
- ***calctime_lapw.pl [SPEED]*** applies equation 4.5 to forecast the runtime of ***lapw1*** in single mode on a given host. In order to yield a machine-dependent calculation time, the program needs the nominal machine speed (e.g. 500) as an argument.
- ***memory_lapw.pl*** applies equation 4.1 to estimate the memory requirement.

Chapter 5

Proof of Concepts

The principles of operation of **`W2GRID`** as well as the **`WIEN2k`** plugin have been described in detail. To prove, that the concept of the new middleware works for realistic applications, this chapter will provide results from the installation on different hosts and the work with the **`WIEN2k`** plugin. Table 5.1 shows the hosts, which have been chosen as testbeds: The Gnu

	HAL	***ATHENA***	***GESCHER***	***LUNA***	***AURORA***	***AURA***
Platform	AIX	SuSE	SuSE	Solaris	Redhat	SuSE
Perl v.	5.005_03	5.8.5	v5.8.8	5.8.4	v5.8.0	v5.8.6
RAM (GB)	16 (shm)	1-4	2	8	4	2
CPU	IBM Pow.3	Intel P4	Intel P4	AMD Opt.	Intel Xeon	Intel P4
Arch	32bit	32-64bit	32bit	64bit	64bit	32bit
Nodes	3	15	16	72	72	8
CPU/node	16	1	1	2	2	1
cores/CPU	1	1-2	1	2	1	1
queue	Cmd	Cmd	PBS	SGE	LL	SGE
symbol	[A]	[B]	[C]	[D]	[E]	[F]

Table 5.1: Testbed hosts for the proof of principle

C-compiler was available on all testbeds, hence the sources of the wiensql-database (section 2.5) and several tools (section 2.8) were always compiled with *gcc*. The GridClient was installed on the same host, which runs the master of the virtual ***ATHENA*** cluster. If it is not explicitly mentioned in sections 5.1.1–5.1.6, the connection between the GridClient and the GridServer was established with the 'socket' plugin, and for the filetransfer the 'scp' plugin was used.

5.1 Installation of W2GRID on the testbed hosts

5.1.1 HAL

The rather old system IBM RS/6000 SP 9070-550 is equipped with an AIX operating system (version 4.3) and consists of three independent nodes with 16 Power3, 375 MHz, Nighthawk2 CPUs and 16 GB of shared memory on each node. **`W2GRID`** was actually installed on a single node, since the cluster was only used for testing the infrastructure. The installation of the database made several bugs of the original C-code apparent, which had to be fixed. Another difficulty occurred due to the quite old Perl-version, which had several bugs[1]. In contrast to all other architectures the compilation of the modules did not work properly. Therefore the module-installation had to be skipped and instead the default-library was used, which is provided as a second choice for such cases. Furthermore this Perl version failed to properly send and receive encrypted strings, although the AES encryption and decryption alone worked as desired. Consequently the encryption was turned off (see section 3.2.1). The installation of **`W2GRID`** succeeded and the resulting infrastructure was operable, but it showed a noticeable latency with all RPC-commands due to the module replacement.

5.1.2 ATHENA

An array of different local desktop computers was used to build a virtual **`W2GRID`** cluster 8 2.1 as illustrated in Fig.2.1. The desktops run different versions of SuSE Linux operating systems (9.1 to 10.1) and use Intel Pentium IV processors ranging from old 32bit 1.7-2.55 GHz to 64bit Core Duo 2.4 GHz. The computers are equipped with 1-4 GB of RAM. All GridServer-slaves employed the commandline plugin, while the master is not used for calculations (see ’adding slaves’ in the usersguide [97]). No problems are reported for the installation of the individual GridServers. Since the GridClient and the GridServer master are installed on the same computer, the ’cp’ plugin could be used.

5.1.3 GESCHER

This is a small 16-node Beowulf cluster equipped with SuSE Linux 10.1 and 32bit Intel Pentium IV CPUs (3.06 GHz). No problems were observed during the installation.

[1]http://perldoc.perl.org/perl572delta.html

5.1.4 LUNA

The SUN X4100 72-node cluster is equipped with four dual core AMD 64 bit Opteron 275 2,4 GHz processors per node and 8 GByte of memory, it is managed by the SGE. Solaris is used as operating system. Some minor problems were experienced when compiling the database. These problems resulted from an improper statement to free allocated memory, which has been fixed. Additionally the installation scripts triggered an error which has not been observed on other operating systems and could be related to an incompatible flag for the GNU ***grep*** tool. Since this flag '-e' was not recognised by the C-Shell version of this tool, the interpreter was changed to ***tcsh*** for all scripts beyond the basic installation, namely **GRIDSRV** and **GRID-CLIENT** (see section 3.1), whereas the rarely occurring statements in the scripts for the layers **GRIDSRC** and **WIENSQL** were replaced by the corresponding GNU tool ***egrep***, which exists on all platforms so far examined[2]. After this fix, the installation and configuration of the Perl-daemons could be completed without any difficulty. Since the host had been commissioned just a few weeks before this thesis was submitted, the time for debugging and thorough testing was too short. Alas, there are no long-term stability proofs available so far.

5.1.5 AURORA

The IBM 1350 cluster, which uses the LoadLeveller for load-management is equipped with two Pentium IV Nocona 3.6 GHz processors on each node and 4 GB of RAM. Problems occurred after the installation of the wiensql-database, since the original code was ANSI- but not POSIX-compliant, hence the signal-statements, namely SIGCHLD resulted in two lines of error-messages being appended to the file /var/log/messages†. This was more annoying than critical, but as a consequence led to an improvement of the code. The new version detects, whether the host is POSIX-compliant or not, and use the respective library. By default the POSIX-library is needed, since it provides advanced methods for signal-handling, which are not available in corresponding ANSI compliant code.

5.1.6 AURA

AURA is a small Beowulf cluster with 8 nodes, each equipped with one 3.2 GHz Pentium IV CPU. The account has been provided by the Photonics-group. Since the account has been granted for testing the infrastructure, it was not used to run **WIEN2k**-cases. This host was especially interesting because it employs the same queuing system (SGE), which is used on

[2]If this does not prove to be applicable for other platforms, all ***grep*** statements will be separated from the code and replaced by a C-Shell script (e.g. ***w2grep.csh***), which can be adjusted to the needs of the actual platform by an automated routine as a fi rst action to be performed by the installation scripts.

LUNA, too. The installation and configuration could be accomplished without difficulties. For the safety of the environment, SSH-tunneling was employed.

5.2 Proof of the standalone principle

According to the principle, described in the introduction (section 1.4) **`W2GRID`** should be installed without the need of additional libraries or tools apart from those, which can be expected to exist on any common Unix/Linux system. Since it was possible to install and configure the infrastructure on each of the testbed hosts the few requirements (i.e. C-compiler, Perl interpreter, C-Shell) were shown to be sufficient. All of the difficulties (described in section 5.1) were due to bugs in the C- and csh-code and do not contradict the standalone principle, since they could all be solved without having to make use of other software than the one provided within the **`W2GRID`** sources. None of the problems reported for the individual machines limited the use. Even on the AIX host, the situation could be handled by making use of several already provided and built-in features of **`W2GRID`**, namely to replace the inoperative module with the default-library and to turn off the encryption. Usually the daemons are only shut down on purpose or upon rebooting the host. Since only the wiensql-database daemon records the accesses from its clients, a sample output of a summary of the logfile content as obtained from the program ***logalizer.pl*** is shown in Fig.5.1.

```
hschweifer@athena:/athena/hschweifer> logalizer.pl
LOGFILE contains 1092676 lines
LOGFILE is 91946208 bytes in size
SHUTDOWN : 15 times
LASTSHUT: 12:30:40 03-07-2006
STARTUP : 16 times
LASTSTART: 07:48:42 04-09-2006
ACCESSES SINCE LAST START:199020
LASTCLIENT at 09:33:21 06-12-2006 [pid:7659]
CODE 0  (everything all right)        : 198953
CODE 1  (unspecified problem)         : 0
CODE 2  (something internal ...)      : 0
CODE 3  (too many clients)            : 0
CODE 4  (IP not acceptable)           : 0
CODE 5  (wrong protocol)              : 0
CODE 6  (wrong key)                   : 0
CODE 7  (login failed)                : 0
CODE 9  (login connection timeout)    : 0
CODE 10 (empty string read)           : 1
CODE 11 (empty encrypted string read) : 6
CODE 12 (write failed)                : 0
CODE 13 (encrypted writing failed)    : 0
CODE 14 (timeout, after some minutes) : 60
CODE 20 (Segmentation fault)          : 0
```

Figure 5.1: The number of connections served by the wiensql-daemon on **ATHENA** and their exit-codes

5.3 Plugin concept

The concept of core and plugins (see section 2.9) allows **W2GRID** to separate different aspects of an architecture into independent pieces. Hence, the interaction can be expanded for all kinds of operating systems, job-submission schemes and to different methods of filetransfer and connections can be expanded. The development and improvement was done based on the experience gained from various architectures. Therefore it was necessary to prove the portability to different hosts.

5.3.1 Connection and filetransfer plugin

Both plugins are routinely applied and were fully functional on all testbed hosts. ***AURA*** was contacted by means of the SSH-tunnel plugin, which always has to examine whether the tunnel is available or not before actually opening the connection, thus leading to a minimal latency, especially if the tunnel was gone and had to be restarted. The different filetransfer plugins implemented so far are all based on the same principle and worked as expected. It was shown, that ftp plugins can be implemented for all connection-less methods (e.g. ***globus-url-copy***).

5.3.2 Platform

When installed on a Redhat System, the former 'Linux' platform plugin was subdivided into 'SuSE' and 'Redhat', since both operating systems differ in certain flavours, which became apparent during the tests. The 'Linux' has been kept though, and is intended to be used as a general platform plugin for those cases where no specialised version is available. Since it is written in a way to serve most kinds of Linux systems, it may suffer from a longer latency due to more complicated regular expression operations. Whenever a new plugin for a Linux-type platform is created one should try to integrate the methods into the standard 'Linux' plugin.
While the AIX and Solaris plugin could only be used on a single host (***HAL*** and ***LUNA***), those for SuSE (***ATHENA*** and ***GESCHER***) and for Redhat Linux (***AURORA*** and ***AURA***) were installed on two hosts each. They proved to be fully portable. To probe the functionality, a tiny test command **{system.test}**$_S$ has been written, which invokes the plugin-functions and displays their results, which can be checked manually by retrieving the values manually. It was successfully shown for all hosts, that the platform plugin worked as desired. An output of this command for ***LUNA*** is shown in Fig.5.2.

```
>system.test
Local hostname:’mgt001’
&nslookup returns:
   Local IP      ’192.168.20.254’
   Hostname      ’mgt001’
&read_local_ip   192.168.20.254
PID of this process:7165
&exists_pid      1
&process_memory  5804
&syspsf returns:
   PID           7382
   OWNER         hschweif
   PARENT        7165
&systop returns 139 processes:
   PID           7165
   USER          hschweif
   MEM (%)       0.0
   CPU (%)       0.0
   COMMAND       gridsrvd.pl
DAEMON PID       27079
&childprocesses  27079 27080 27168
&getuptime       7:39pm up 25 day(s), 8:08, 12 users, load average: 0.04, 0.02, 0.01
&getload         0.02
&totalmem        8064.0
&freemem         7.0
&current_dir     /export/aurora/hschweif/WGRID/demons
&get_home        /export/aurora/hschweif
CURRENT DATE     ’Fri Dec  8 19:39:52 CET 2006’
&read_time       19:39:52 8-12-2006
```

Figure 5.2: The results of the command **{system.test}**$_S$ on host ***LUNA***

5.3.3 Execution plugin

The key issue of the **`W2GRID`** concept is the independence of the execution plugin from the underlying platform. Based on the fact, that the SGE plugin works for ***AURA*** (SuSE) as well as for ***LUNA*** (Solaris), it can be concluded, that independence can be taken granted. Furthermore the ’commandline’ being the default plugin was tested on all six hosts and gave the expected results. Due to the fact, that this plugin invokes the processes on the frontend of a managed cluster, no real calculations have been run there, but only short sleep-jobs. To probe the functionality, **`W2GRID`** provides a test script **{test.exec}**$_S$, which allows to query the load of the queuing system, submit a job (sleep 100), check its status and remove it again. It was applied properly on all hosts. A screenshot of these results is shown for ***HAL*** in Fig.5.3.

```
>test.exec submit 1
 42574
>test.exec exists 42574
 SUCCESS:Process-ID (or Job-ID) ’42574’ exists
>test.exec kill 42574
 SUCCESS:Process-ID (or Job-ID) ’42574’ was killed
>test.exec exists 42574
 FAILED:Process-ID (or Job-ID) ’42574’ does not exist
```

Figure 5.3: The results of the command **{test.exec}**$_S$ on host ***HAL***

5.4 Applicability for other scientific programs

W2GRID is structured to be a more or less general-purpose infrastructure, which fits the needs of different applications. It was already said in the introduction, that **W2GRID** was not exclusively used for **WIEN2k**, but also for an other scientific applications, namely **MCTDHF**. Since the plugin has been developed by the Photonics group [13], only the ideas and the concept will be shown in this section.

5.4.1 Problem definition

The Photonics group develops and uses different numerical simulation codes, such as **MCTDHF**, **TDSE**[3] and **QDS**[4]. To run these codes on their quite heterogeneous computing infrastructure causes -similar to **WIEN2k**- a significant administrative overhead, especially with the parallel **MCTDHF** version [16]. An additional challenge for the researcher is to keep track of the numerous material science cases, which are already computed or are still investigated, since the applications are not only used to compute the results but also to visualise the data and render them e.g. into movies.

5.4.2 Solution

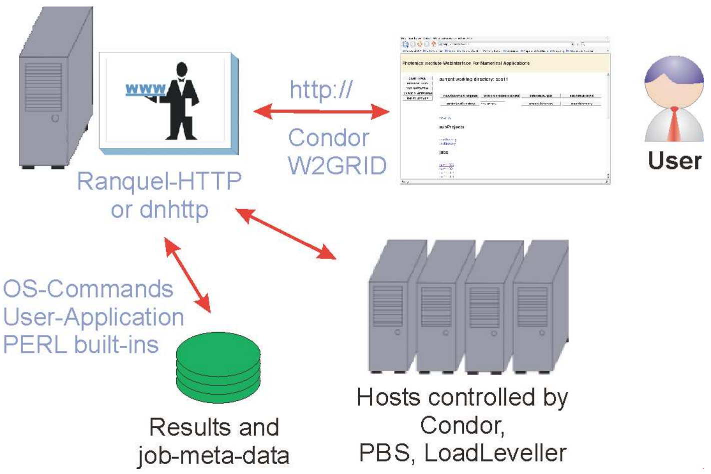


Figure 5.4: Purpose of the web interface for **MCTDHF** and other codes of the Photonics group, figure taken from [91]

A web interface [91] has been created, which serves as a tool for starting calculations, retrieving their status and for checking the output. The results and the numerous output files can

[3]Time Dependent Schrödinger Equation Solver: A single CPU application for doing studies of the dynamics of single quantum mechanical particles.

[4]Quantum Dot Solver: A single CPU application for studying quantum dots.

be handled conveniently and features are provided for the researcher to analyse the data. **W2GRID** is employed for the load-management system, namely the execution and control of processes on a subset of the resources. The web interface does not exclusively employ **W2GRID** but may in principle use any kind of infrastructure (e.g CONDOR [58], VGE [78]), as long as an interface can be implemented. The principle of the web interface is illustrated in Fig.5.4. **W2GRID** is used as a batch system for hosts, for which no root-access is available to install CONDOR, whereas the latter is used for all the others. It was shown, that the same mechanisms, which apply for **WIEN2k** also worked for another application, that needs only the basic mechanisms of distributed computing like filetransfer, job-start and the respective monitoring. Figure 5.5 shows the testbed for the web interface, where **W2GRID** comes to

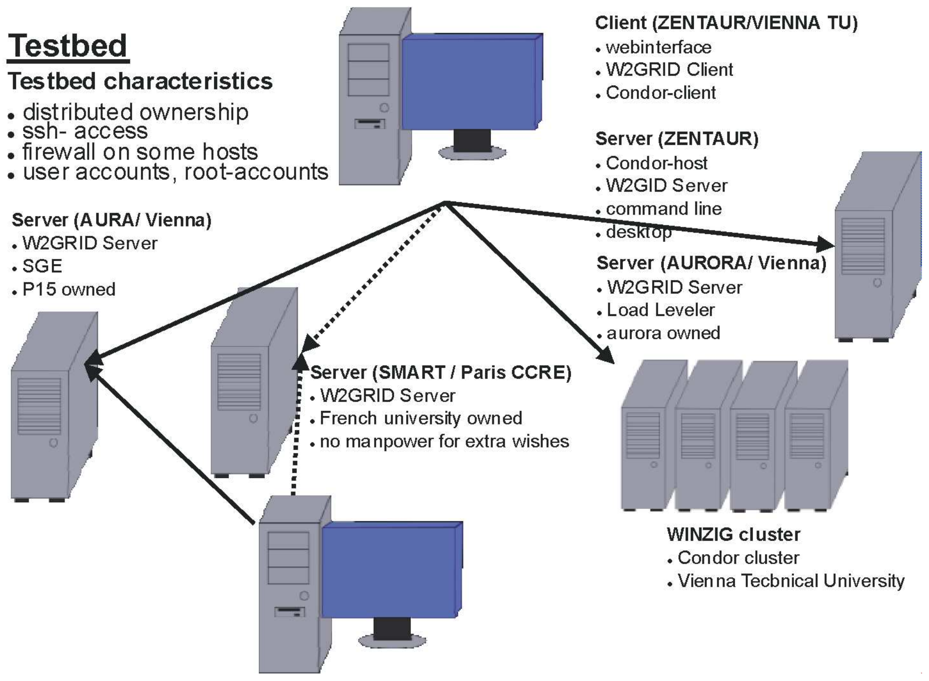


Figure 5.5: The testbed for MCTDHF at the Photonics institute, fi gure taken from [91]

play at several locations. Some of the hosts can only be reached by SSH-tunnels to avoid compromising remote security.

5.5 WIEN2k

In chapter 4, the **WIEN2k** application plugin was presented in terms of its functionality and the details of the operation principle. This section provides data and results from realistic examples of **WIEN2k**-CASES, which have been run on **W2GRID**. Certain data are extracted from the logfiles to confirm the assumptions made in chapter 4.

5.5.1 Portability of the plugin

The **WIEN2k** plugin was successfully installed on all hosts in the way as described in section 3.3.5. The $WIENROOT environment variable had to exist and must point to the proper location of the **WIEN2k** executables, otherwise the installation would have been cancelled[5]. All parts of the plugin are fully portable to all hosts of the testbed.

5.5.2 Host selection

A central capability of the plugin is the ability to make an autonomous decision on the proper resource, which best suits the requirements of a CASE with respect to its memory consumption and estimated computing time. This basically needs to be done in two steps. A static one, which offers a match-making of the requirements against the contents of the registry and a dynamic one, which queries each host and analyses its current load and memory situation.

5.5.2.1 Memory constraints (static)

In order to prove that the middleware is able to make a proper pre-selection, a set of test CASES was created[6], which differ only with respect to their memory requirement. To prove the concept it was sufficient to fetch the proposals for the .machines†-file by the aid of the command **{wien.mkmachines}**$_C$ instead of actually submitting the jobs to **W2GRID** by **{wien.exec}**$_C$ (see table 5.2)

	I	***II***	***III***	***IV***
M	9000	11000	15000	22000
n_{at}	3	3	3	3
k	6	6	6	6
c	1	1	1	1
size (MB)	1620	2420	4500	9680
hosts	[A-E]	[A][B][D][E]	[A][D][E]	[A]

Table 5.2: Different memory requirements of a **WIEN2k**-CASE as selection criteria for the GridClient

The *size* was calculated according to equation 4.1 by using the tool ***memory_lapw.pl***. The obtained host selections agree with the expectation, since only those computers were proposed, which have sufficient memory to run the CASE. Furthermore it should be noted, that host ***AURA*** [F] was never considered, because it did not have the **WIEN2k** plugin installed. Since

[5]**WIEN2k** has already been pre-installed on all machines.
[6]basically the same CASE with different MATRIX-sizes.

the GridClient keeps in its registry a list of installed programs for all GridServers , this fact is part of the static host selection[7].

5.5.2.2 .machines†-file proposals and dynamic selection criteria

To check the ability of **`W2GRID`** to parallelise a CASE properly, the input as shown in Table 5.3 was used, which was adjusted in such a way that all hosts are acceptable. The parameters

	si3n4
M	923
n_{at}	3
n_{LM}	112
k	153
c	1
$size$ (MB)	17

Table 5.3: Parameters of a prepared **`WIEN2k`**-CASE, which has been used to analyse the .machines†-fi le proposals of the individual hosts.

that allow a user to influence the .machines†-file proposals are 'mintime', 'maxhosts' and 'cpulimit', which by default are set to the following values: 'mintime=120', 'maxhosts=unlimited' and 'cpulimit=66' respectively. By the use of the command **{wien.mkmachines}**$_C$ and keeping these default-parameters the results shown in table 5.4 were obtained, which are discussed in the following subsections. The column 'LAPW1 runtime' represents an average runtime for a single LAPW1-chunk (which is equal to the runtime of the whole LAPW1 workflow-step, according to Fig.4.3).

	HAL	***ATHENA***	***GESCHER***	***LUNA***	***AURORA***
free nodes	16	15	16	2	2
queued nodes	0	0	0	0	68
load	100%	20%	0%	97%	97%
No. nodes	5	5	6	2	2
No. processes	5	5	6	4	4
LAPW1 runtime (s)	317	138	142	134	150
proposal	2*30+3*31	39+32+30+29+23	3*25+3*26	39+3*38	39+3*38

Table 5.4: .machines†-fi le proposals

[7] All informations, the GridClient stores in its registry is shown for ***GESCHER*** in Fig.A.20 in the appendix

- ***HAL***: Since the commandline plugin is used, all 16 CPUs can be employed for the calculation although they may be occupied already. A heavy load of 100% is indicated, but this is irrelevant to the GridClient and just serves as an information. As 100% is the upper limit that is displayed, the 'true' load is concealed, which had a value of 19 ($\sim$118%). Up to five additional processes (the effective load will be 24, which equals $\sim$150%) will yield a single processor utilisation of more than 66%, whereas the sixth process causes the efficiency to drop below the threshold. Therefore the plugin may exploit a maximum of five CPUs and distribute the workload as evenly as possible.

- ***ATHENA***: The constraint for the selection was the runtime of a single chunk (120 sec). If a sixth host had been added, the runtime of one of the chunks would have dropped below the limit. The different load-distribution is due to the different powers (speed number) and loads of the machines.

- ***GESCHER***: The selection is due to the mintime constraint.

- ***LUNA***: The host had got two free nodes. It would employ both for this job and run two tasks per node with two threads, which makes four processes and therefore four entries to the machines file. So far the **WIEN2k** plugin does not quantify the effect of the threading in the proposal, but the job submission does. The runtime is still calculated on the basis of a single thread and should be less than 134 seconds.

- ***AURORA***: The cluster was occupied and 68 nodes were already requested, and thus it is unlikely, that the job will be run soon. The proposal uses two nodes, and two tasks per node, which again makes four processes and four entries to the .machines†-file. The threading is not used, therefore the runtime is accurate.

5.5.3 Results obtained from realistic CASES

After it has been shown, that **{wien.mkmachines}**$_C$, the binaries of the plugin and all required libraries work, realistic CASES have been run. The workflow of this command and some of the triggered processes are shown in Fig. 4.9 on page 134. For this purpose the three CASES shown in table 5.4 have been submitted to all hosts ([A-E]). To bypass the automated scheme, which decides on the optimal resource, the constraint to run on a dedicated server (option '-server *[SERVER]*') was used[8]. The runtime $time_{LAPW1}$ is an average runtime for LAPW1, which has been calculated for an Intel Pentium IV 2.4 GHz (speed=500).

[8]Otherwise it would have been diffi cult to study the behaviour of all hosts, if the selection is left to the GridClient.

	cu2o	***si3n4***	***ir_7l***
M	734	3813	1924
n_{at}	2	3	4
n_{LM}	11	112	39
k	165	6	272
$size$ (MB)	5	290	37
c	0	1	0
$time_{LAPW1}$ (s)	124	1906	3627

Table 5.5: Three different realistic cases, which have been run on all hosts.

5.5.3.1 Analysis: Self-benchmark

After an SCF calculation has come to an end without observing an error, the GridServer (master) extracts the runtimes of the machines as well as the number of k-points they processed and invokes the command **{wien.benchmark}**$_S$, which compares its own prediction against the actual runtime and modifies the factor g accordingly by ways of equation 4.6. On a virtual host, the GridServer master executes **{wien.benchmark}**$_S$ on all of its slaves, whereas on a managed cluster the standalone GridServer benchmarks itself in the same way by establishing a connection to its own port. In order to study this scheme, one of the GridServer slaves on host ***ATHENA***, namely 'ne' has for testing purposes been set up with a completely wrong machine-speed (1500 instead of 800), which led to an over estimated proposal for this machine. The logfile shown in Fig.5.7 illustrates, how the initial k-point distribution suffered from this wrong machine-speed of host 'ne', whereas the following cycles already used a much better distribution. The employed scheme can only effectively change the distribution after the first cycle has been completed, otherwise the pattern matching does not return a result from the dayfile. A few lines of the logfile content are shown in Fig.5.6. The original value of

```
1: (INFO) found in dayfile: HOST 'ne'
2: (INFO) nslookup resulted in IP '128.130.134.32'
3: (INFO) host 'ne' was identified as 'ne' in the database
.   ...
4: (INFO) benchmarking host 'ne'
5: (INFO) Latest Transmission was considered type:4 (everything >=0 is good)
6: (INFO) sending request 'wien.benchmark' to gridserver (parameter omitted)
7: (INFO) 'ne' said: "speed diluted by '0.4149' from '1.0000' to '0.9464'
```

Figure 5.6: Selected content of a logfi le, indicating the self-benchmark

the correction-factor g has been 1. The results of this calculation show, that the relation of estimated and real runtime yielded a factor of 0.41. The self-benchmark does, however, not allow such big leaps in the correction-factor with a single run, furthermore it must be avoided, that short CASES have too much influence on the result. This is why, the value for g was only reduced by roughly 5.4%. Numerous runs are necessary to achieve an accurate prediction (line 7).

5.5.3.2 Analysis: Adjusting the k-point distribution

So far this feature has only been implemented for a virtual cluster such as ***ATHENA***, since it can be assumed that often usual HPC clusters will consist of identical nodes. During the execution of a CASE the k-points are redistributed as a reaction to a changed load-situation or because the initial guess of the machine-speed has been wrong. The redistribution uses the actual runtimes, which are extracted from the logfile CASE.dayfile† and represent the real performance in contrast to the proposals of the individual hosts, which is based on the runtime model (see equation 4.5) and may not be accurate. This scheme is illustrated with one CASE of table 5.5, namely ***ir_7l***.

```
 1:     ne  (101) 441.319u 5.728s 7:37.52 97.7%
 2:     eos  (48) 212.749u 2.368s 3:37.42 98.9%
 3:     xe   (46) 193.476u 2.168s 3:18.20 98.7%
 4:     kr   (44) 184.767u 2.152s 3:11.22 97.7%
 5:     iris (33) 252.735u 4.652s 4:23.85 97.5%
..      ...
 6:     ne   (58) 255.579u 3.060s 4:25.21 97.5%
 7:     eos  (59) 262.584u 3.116s 4:31.09 98.0%
 8:     xe   (61) 267.500u 3.132s 4:34.70 98.5%
 9:     kr   (61) 260.360u 2.960s 4:28.35 98.1%
10:     iris (33) 252.395u 4.604s 4:23.74 97.4%
..      ...
12:     ne   (58) 261.220u 3.316s 4:34.63 96.3%
13:     eos  (59) 264.192u 2.924s 4:35.39 96.9%
14:     xe   (60) 265.908u 3.088s 4:32.58 98.6%
15:     kr   (61) 266.096u 2.924s 4:32.66 98.6%
16:     iris (34) 263.576u 4.852s 4:50.67 92.3%
```

Figure 5.7: Selected content of a CASE.dayfi le, which shows the automated adjustment of the k-point distribution

The CASE was run with a smaller MATRIX size (1418 instead of 1924) in order to obtain the results faster[9]. The principle is demonstrated by manipulating the speed number for host 'ne'. The same run was used to demonstrate the self-benchmark. The few important lines have been extracted from the dayfile (ir_7l.dayfile†) and are shown in Fig.5.7. While the k-point distribution is unfortunate in the beginning (line 1), it is already close to optimal in the following cycle (line 6), where host 'ne' received almost half the k-points it originally processed. The relation of the k-points 58/101 (0.57) is close to the relation of the speed-numbers 800/1500 (0.53).

[9]The runtime-prediction is cut by half to 1494 seconds.

Chapter 6

Discussion

It was shown in chapter 5, that it is possible to install the middleware on many typical operating systems, which are used for scientific computing resources. All reported difficulties of the installation routines (csh-scripts) and the C-code (wiensql-database), which were experienced on some hosts of the testbed could be solved without a violation of the standalone concept, since the fixed code afterwards still works on all the other target systems. Third-party software was not required in either case. It is expected that upon porting the code to other platforms (such as HP-UX, IRIX, Tru64, Ultrix, Mac OS X, NetBSD, NextStep, Plan 9, QNX, System V -to mention a few Unix-like systems, among which some are proprietary and not 100% POSIX compliant-) may lead to similar difficulties, which will be solved in the same way.
The idea to use a standardised scripting language proved to be successful, as Perl provides powerful tools to solve complex problems. Difficulties were only experienced with the AIX operating system, but those were related to bugs in the rather old Perl interpreter and could be solved too. Since it is expected that all newer hardware will be equipped with more advanced Perl versions than 5.005, these incompatibilities are unlikely to recur. Although the set of investigated platforms has been limited to a few essential ones, it could be shown that the principle of operation works as desired. The approach of **`W2GRID`** to separate all platform and job-submission related tasks as plugins from the rest of the infrastructure (core) as well as from the scientific applications provides the desired framework to run scientific applications such as **`WIEN2k`** or **`MCTDHF`** on different kinds of hardware with the same application plugin. It was shown, that the platform and the execution plugin are independent of each other (see section 5.3.3) and that the application plugin may also be written in an independent way such that it does not require to call platform or queuing system specific commands explicitly. Hence it can be concluded, that additional platforms, queuing systems and applications can be integrated into **`W2GRID`** simply by writing the corresponding plugins. A guide for this development is given with this thesis (see chapter 3).

The concept of lightweight middleware as presented in the introduction (see section 1.3.2) imposes certain limitations for the design of the application plugin, as it sacrifices certain features, which are rarely required in scientific computing and instead focuses on the few essential issues of distributed computing for the sake of portability and a better usability. For example **W2GRID** is well suited to run programs within a single domain having a shared filesystem, whereas it is not possible to parallelise programs across several computing sites, since parallelism cannot be added by **W2GRID** but must already be provided by the application. If a calculation is started on a remote host, which does not have access to the local filesystem **W2GRID** can manage the filetransfer in both directions. This updating of input files on the GridServer sites[1] and the output files on the local site is done in regular intervals and therefore comes with a certain latency, which does not allow immediate steering, since one does not possess real time data. Furthermore the filesystem is used as the primary data-storage. Certain informations of a maximum string length of roughly one Megabyte may also be stored in the wiensql-database, whereas extended database-support such as interfaces to mysql or postgre-sql or even distributed data-storages as provided by data grids cannot be offered by **W2GRID**. Therefore programs, whose files must be up-to-date all the time or have to employ remote databases are better served with other middleware. **W2GRID** is already used by the Photonics group to run the **MCTDHF** and **QDS** codes on their computing resources (see Fig.5.5).
Already at the beginning of the development phase, security has been taken seriously, since it was evident, that such kind of middleware may put the safety of a computing centre at risk if it is vulnerable to the most basic kinds of attacks. For this reason the AES encryption has been employed in order to protect the daemons and to provide a sufficient security, such that system administrators do not consider their firewalls punctured from the inside. **W2GRID** proved to work well even in critical environments, where the computing resources could only be contacted by the aid of SSH-tunnels with certificate-based authentication.
The present implementation of **W2GRID** requires, that a GridServer daemon runs on the frontend of each computing site and is accessible from the outbound network. Restrictions imposed by firewalls can be addressed properly with SSH-tunneling, but there is the policy enforced by some system administrators, who will not give users the permission to run daemon processes at all, due to security concerns. This problem can be overcome by operating the host in question remotely through an interactive shell-session. At the moment such hosts cannot be used, unless this feature is implemented in **W2GRID**. Furthermore interactive access is required.
Many Grid-infrastructures, also lightweight ones [78] highlight the importance of quality of service (QoS), because certain distributed applications (e.g. medical data processing) need

[1]for the purpose of steering

reliable forecast models. This issue has largely been neglected by **W2GRID**, which is due to the fact, that the infrastructure is supposed to be operated under user-permissions. **W2GRID** may only guarantee the kind of quality of service, which is provided by underlying queuing system, but is not able to enforce a strict policy for the resource usage by its own means.

The **WIEN2k**-application plugin proved to work on all testbed hosts as it delivered the expected results. Although the runtime forecasts on all hosts have been in the proper range, the analytic performance model as applied at present is too rigid to be used on all platforms, because the parameters (α, β, γ) depend on numerous parameters, which cannot be attributed by simply adjusting a linear factor. It was observed, that the model applies well for 'known' systems such as the Intel architecture, but does not scale accordingly on Power 3 processors. Instead of improving the forecast by modifying a single linear parameter it may be better to tune α,β and γ with each benchmark as this will provide a more reliable prediction model in the future.
The automated adjustment of the k-point distribution showed to enhance the CPU efficiency significantly since this is done with respect to the dynamically changing load situation of the hosts and allows to optimise the utilisation of resources on a given computing site. Although this scheme is well suited for cases with many k-points, it became evident that it does not work well for CASES with fewer, since **W2GRID** usually shifts excess k-points from a slower host to the faster host (with the least runtime for its chunk) without caring about the runtime difference, which may result in an unfavourable load balance. This scheme needs to be improved in order to optimise the k-point distribution with respect to an optimal total-runtime of the involved chunks. Additionally the method cannot be applied unless a whole iteration is completed, since the regular expression pattern does not work otherwise. If the job $\ulcorner\!\llcorner wien.exec \urcorner\!\lrcorner_S$, which checks the calculation and optimises the k-point distribution happens to be run by the controller in exactly that time-interval between the finishing of a preceding iteration and the start of the parallel LAPW1 in the next one, already the second iteration may profit from an improved distribution, otherwise it is usually the third iteration. It is evident that the mechanism must incorporate a second pattern, which allows to extract the runtimes already after the parallel LAPW1 step is completed.

It can be concluded that **W2GRID** turned out to be flexible in the desired way. The infrastructure can be installed with the means and permissions granted to a user and is relatively simple to configure. It can be used on hosts, where many other middleware products fail due to insufficient permissions or unavailable libraries. With the development of new plugins for other platforms, queuing systems, filetransfer- and connection-methods it can in principle be ported to any Unix-like computing resource.
As it was shown, the offered solution also applies to other programs. The effort for the devel-

opment of such application plugins scales with the desired capability. While a simple workflow may be already satisfied with some hundred lines of Perl-code (including the processing of input arguments and the recommended documentation) complex tasks like those performed by the **WIEN2k** plugin may need up to several thousand lines of code. This is due to the fact that many features, which are needed in a similar way by different applications are already offered as simple function calls that allow to execute complex tasks like the filetransfer by a single line of code. On the other hand all special features exclusively needed by the application such as the sophisticated distribution of k-points on the available nodes in the case of **WIEN2k** have to be coded explicitly and thus require the same effort of program code as with any other language or Grid middleware.

One of the most promising features of **W2GRID** is that it allows application developers to develop a single plugin for their programs, which may in principle be used on many different architectures, regardless of what kind of queuing system may be installed there. This feature is of course limited to the capabilities, which are supported by the plugin-type (e.g. The execution plugin supports MPI but not PVM). Since execution plugins may also be written for heavyweight middleware such as CONDOR or GLOBUS, **W2GRID** could be used as a top-level batch system and extended to larger scales. Many features, which have been omitted in the initial design of the middleware can be added as plugins later on. **W2GRID** in its current state (version 2.8.11) is ready for use.

Abbreviations and special terms

Abbreviations

- **AES** Advanced Encryption Standard (see below)
- **AHE** Application Hosting Environment
- **AJO** Abstract Job Objects
- **API** Application Programming Interface[2]
- **CLI** CommandLine Interface
- **CPAN** Comprehensive Perl Archive Network
- **DES** Data Encryption Standard
- **DFT** Density Functional Theory
- **GRAM** Grid Resource Allocation Management
- **GSI** Grid Security Infrastructure
- **GUI** Graphical User interface
- **HF** Hartree Fock
- **HPC** High Performance Computing
- **HTTP** HyperText Transfer Protocol
- **HTTPS** HyperText Transfer Protocol Secure
- **IP** Internet Protocol
- **LCAO** Linear Combination of Atomic Orbitals

[2]http://en.wikipedia.org/wiki/API

- **LHC** Large Hadron Collider
- **LL** LoadLeveller
- **MCTDHF** Multi Configurational Time Dependent Hartree Fock
- **NIST** National Institute of Standards and Technology
- **PBS** Portable Batch System
- **PID** Process-identifier (see below)
- **PVM** Parallel Virtual Machine
- **QDS** Quantum Dot solver
- **QN** Quantum Number
- **QoS** Quality of Service
- **RPC** Remote Procedure Calls (see below)
- **RSL** Resource Specification Language
- **SCE** Scientific Computing Environment
- **SDK** Software Development Kit
- **SGE** Sun Grid Engine
- **SGML** Standard Generalised Markup Language
- **SGS** Styx Grid-Services
- **SOAP** Simple Object Access Protocol
- **SQL** Structured Query Language
- **TCP** Transmission control Protocol
- **TDSE** Time Dependent Schrödinger Equation Solver
- **UNICORE** Uniform access to Computing Resources
- **UPL** UNICORE Protocol Layer
- **VGE** Vienna Grid Environment

- **VGCE** Vienna Grid Client Environment
- **VGSE** Vienna Grid Service Environment
- **W2GRID** Wien-to-Grid
- **WEDS** Web-Service based Environment for Distributed Computing
- **XML** Extensible Markup Language

Special notations

- $\ulcorner JOB \urcorner_{S,C}$: A background process. The attached suffix identifies, for which daemon it is written.
- ♯directory/: The directory is marked with a preceding sharp (♯) symbol and written in 'Avant Garde'. In order to avoid scrambling the text with long paths, only relative pathnames, which originate in ♯$GRIDSRC/ will be used. For example ♯libs_perl/ is to be expanded into ♯$GRIDSRC/libs_perl/. However, if a path is located outside of ♯$GRIDSRC/ (e.g. ♯$GRIDROOT/) it will be specified explicitly.
- ***executable***: Executables are written in 'Adobe Times' (bold and itallic).
- **PROGRAM**: Tradenames for programs such as **WIEN2k** or **MCTDHF** are written monospaced.
- file†: Files are written in 'Avant Garde' and marked with a succeeding dagger. If not said otherwise in the text, ♯$GRIDSRC has to be added to the path.
- **(column)**: In the chapters 2–4 numerous column-names are used and explained. To improve the readability, the column is written bold and enclosed in parentheses.
- **{COMMAND}**$_{S,C}$: The RPC-command, which can be invoked on the daemon. The suffixes serve the same purposes as for the $\ulcorner JOBS \urcorner$.
- ***PID***$_Q$: The process-ID, which is returned from a queuing system. This is not identical to the procecess-ID's as obtained from the commands ***ps*** or ***top***.
- $VARIABLE: An ordinary variable (e.g. STRING, INTEGER, FLOAT, ...). Perl handles the types internally, type-definitions are therefore not required.
- @ARRAY: The array data-type of Perl. Arrays may only contain numbers as indices (e.g. $ARRAY[0] is the first element).

- \@ARRAYREF: A pointer to an array.
- %HASH: A special kind of array in Perl. The indices, which point to its content do not necessarily have to be numbers but may also be strings (e.g. $HASH{0} or $HASH{a}) Different to the square brackets used for the conventional arrays, the HASH uses curled braces. Its content may be other hashes, hence this variable type is useful to simulate objects.
- \%HASHREF: A %HASH cannot be passed to a function, due to the intrinsic way how Perl handles the input. Therefore it must be passed as a reference, which is nothing else than a pointer to the location in memory, where the content of the HASH is stored. The value of this reference variable is a string and can also be printed to the screen. It will show a hexadecimal address space. Assuming $ref=\%HASH, the original value can be retained by dereferencing the pointer with %HASH=%{$ref}
- BUTTON: A button on the keyboard.

Special terms used in this thesis

- **Workflow:** terms the movement of tasks through a work process. More specifically, a workflow is the operational aspect of a work procedure: how tasks are structured, who performs them, what their relative order is, how they are synchronised, how information flows to support the tasks and how tasks are being tracked. It is a mostly visualised abstraction of some processes[3].
- **RPC:** Remote-Procedure-Calls are widely used in distributed computing and allow to invoke commands remotely on different hosts. RPC's make the construction of distributed programs an easy task by extending the conventional procedure-call scheme to distributed environments and hiding from programmers the complications involved in concurrent communication, as well as transmission errors. When writing PRC programs, programmers can use paradigms similar to conventional local procedure calls while calling and called procedures are allowed to reside on different machines. In the RPC mechanism, when a caller makes a remote procedure call, the caller is suspended and the parameters of the called procedure are passed across the network to the callee where the execution of the called procedure takes place. After completion the result is passed back to the caller, which resumes its execution as if it returned from a local procedure call. this paradigm is well suited for use in the client-server model. [104]

[3]http://en.wikipedia.org/wiki/Workflow

- **Registry:** Always refers to some kind of data-storage, mostly a database-table. To register an item means to insert data, whereas de-register refers to deleting data from the storage container.
- **AES:** AES is a symmetric key encryption technique which replaces the commonly used Data Encryption Standard (DES).It was the result of a worldwide call for submissions of encryption algorithms issued by the US Government's National Institute of Standards and Technology (NIST) in 1997 and completed in 2000. The winning algorithm, Rijndael, was developed by two Belgian cryptologists, Vincent Rijmen and Joan Daemen. AES provides strong encryption and has been selected by NIST as a Federal Information Processing Standard in November 2001 (FIPS-197), and in June 2003 the U.S. Government (NSA) announced that AES is secure enough to protect classified information up to the TOP SECRET level, which is the highest security level and defined as information which would cause "exceptionally grave damage"to national security if disclosed to the public.The AES algorithm uses one of three cipher key strengths: a 128-, 192-, or 256-bit encryption key (password). Each encryption key size causes the algorithm to behave slightly differently, so the increasing key sizes not only offer a larger number of bits with which you can scramble the data, but also increase the complexity of the cipher algorithm.
- **client/server:** The naming convention results from the dataflow. The server is the process, which listens to a certain port for incoming connections and offers certain capabilities. An other process, the client connects to this port and requests these capabilities. This is especially important in a case, where the GridServer daemon connects to the wiensql daemon. The convention makes the GridServer the 'client' because it is served by the wiensql daemon, which is hereby the 'server'.
- **PID (PID)** Every process on the host has got a unique number, which can be used to identify it. Within the scope of this document, the PID is the numeric process-identifier for commandline-processes. It is not equal to the JOB-ID, which is assigned by a queuing system to a submitted job.
- **CASE** This refers to the name of the **`WIEN2k`** calculation in question and to the name of the directory, which will also be the prefix for most files (see section 1.1.3.4).
- **LAPW1** As a workflow-element, the program is written with uppercase characters, whereas if it is used as a binary, it is written as the respective binary ***lapw1***.
- **XML** The Extensible Markup Language (XML) is a W3C-recommended general-purpose markup language that supports a wide variety of applications. XML languages or 'di-

alects' are easy to design and to process. XML is a simplified subset of Standard Generalised Markup Language (SGML). Its primary purpose is to facilitate the sharing of data across different information systems, particularly systems connected via the Internet. Formally defined languages based on XML (such as RSS, MathML, XHTML, Scalable Vector Graphics, MusicXML and thousands of other examples) allow diverse software reliably to understand information formatted and passed in these languages[4].

[4]http://en.wikipedia.org/wiki/XML

Definition of Grid Computing

The meaning of the term ’Grid computing’ changed over the years [32, 51, 105, 106], and it is quite difficult to find an obliging definition, since developers look at it from their very personal perspective. The present understanding is, that *“a computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities*” [32]. It forms with the pooling of resources such as computing resources and storage capacities, hence the most prominent types of ’grids’ are computational grids and data grids [107]. The key concept is the ability to negotiate resource sharing arrangements among a set of participating parties (providers and consumers) and then to use the resulting resource pool for some purpose. Foster et.al. [106] define a Grid as an infrastructure, which coordinates resources, that are not subject to centralised control, uses standard, open general-purpose protocols and interfaces and delivers nontrivial qualities of service. Cluster management systems such as PBS or SGE do therefore not qualify as grids, due to the centralised control. Other authors [98] determine Grids by their Resource management system, which is governed by tasks such as access to resource information, status monitoring and scheduling, reservation management and execution. Depending on the purpose and the size, it may be of interest to maintain record of the usage for accounting and billing [108]. Yet most of these requirements represent a certain ’common sense’ within the Grid-community, whereas s stringent definition is still outstanding.

Important library functions for development

Only the essential functions are illustrated and explained. A complete list is provided in the developersguide [103]. The informations provided in this chapter require a basic understanding of Perl.

Utilities

The utility-functions are already included by default in any **{COMMAND}** and $\ulcorner JOB \lrcorner$ and provide some general-purpose capabilities.
File: $GRIDSRC/libs_perl/utils.pl†
Perl: include “$PATH{LIB}/utils.pl”;

- **&run_external(*$statement*)**: Executes the string ’$statement’ on the commandline from within an internal Bourne-Shell (sh) and returns the obtained result. The commands must conform to the Bourne-Shell syntax.
- **&rijndenc(*$key,$plain_text*)**: Encrypts a given $plain_text by the aid of the AES algorithm and the supplied $key and returns the encrypted text.
- **&rijndec(*$key,$encrypted_text*)**: Decrypts an $encrypted_text and returns the plain text.
- **&create_tempdir()**: Creates a temporary directory in ♯$GRIDROOT/temp/ b $_{2.5}$ and returns its name.
- **&file_tempname(*$dir,$rump*)**: Suggests a unique filename in directory $dir but does not create it. The returned result is composed of the string ’$rump’ and an attached number.

- **&extract_parameter(*$string*)**: takes a $string of the form ’*a=1 b=10 x=test*’ and returns a hash e.g. %TEMP. The elements of the hash are the parameters contained in the string (e.g. $TEMP{a}=1).
- **&filestat(*$filename*)**:retrieves the current file informations and returns a hash containing the results (e.g. MTIME, ATIME, CTIME, SIZE, ...).

Errors and Warnings

W2GRID keeps an internal stack of error- and warning-messages, which accumulates all messages. The errors and warnings are automatically appended to any result string returned from the daemon, hence the developer does not have to print them explicitly. The respective functions are already included by default in any **{COMMAND}** and ⌜*JOB*⌟.
File: $GRIDSRC/libs_perl/w2grid.pl†
Perl: include “$PATH{LIB}/w2grid.pl”;

- **&write_error(*$errmsg*)**: Adds $errmsg to the error-stack and changes the internal status. &check_error() now returns ’1’. The error-message is also appended to the logfile (see &gridlog__write()).
- **&check_error()**: Returns ’1’ if &write_error() has been called at least once, or ’0’ otherwise.
- **&write_warning(*$warnmsg*)**: Similar to &write_error(). The messages are usually not critical and are appended to the logfile, too.
- **&check_warning()**: Similar to &check_error().
- **&check_abort()**: Returns ’1’ if either an error has been triggered or if the function &input__finish() was called (see Fig.A.4, line 22).

Verbose and regular output

The verbose-strings may be used to debug the code in the development phase, but can additionally serve as an intrinsic documentation of the code. To see these strings on STDOUT, the verbosity of the **daemon** must be turned on explicitly (see section 3.2.1). The functions are already included.
File: $GRIDSRC/libs_perl/w2grid.pl†
Perl: include “$PATH{LIB}/w2grid.pl”;

- **&write_verbose(*$verbose*)**: Writes a verbose message to the stack, which is appended to the logfile,too (see &gridlog__write()).
- **&write_verbose_internal(*$verbose*)**: Writes a verbose message to the internal stack. This message is not appended to the logfile.
- **&write_result(*$result*)**: The most important function for RPC-commands. The string $result will be returned to the client. Different to verbose, warning and error it is not written to a stack but overwritten each time, the function is called. It is recommended to use it only a single time at the end of a command (line 32 in Fig.A.4). The string is not appended to the logfile.

Logfi les

The verbose output may only be sent to STDOUT if the verbosity of the daemon has been enabled and if the terminal still exists. In any other case, the output is usually lost, unless it is written to a logfile, which collects all $strings, obtained from the functions &write_verbose(), &write_error() and &write_warning(). The logfile-functions are available by default to commands and jobs.
File: $GRIDSRC/libs_perl/gridlog.pl†
Perl: include “$PATH{LIB}/gridlog.pl”;

- **&gridlog__open(*$filename*)**: Opens the logfile ’$filename’ or newly creates it if it did not exist. An absolute pathname must be provided.
- **&gridlog__write(*$text*)**: Writes the $text to the logfile. The use of this function should be avoided in favour of the above mentioned ones.
- **&gridlog__close()**: Usually the log-data is not written immediately to the file after each call of &gridlog__write() but instead stored and finally written with the callingo of the function &gridlog__close(). If the script is terminated prematurely for any reason, the respective command will not be reached and the data is lost. Although the logfile will automatically be closed at the end of every **{COMMAND}**/$\ulcorner JOB \urcorner$ by the subsequent cleanup-process of the daemon/controller it is however recommended to include this statement at the end of each command/job in order to avoid the risk of a loss of data.
- **&gridlog__flush()**: immediately writes all accumulated log-data, but does not close the logfile.

Execution plugin

The functions of the execution plugin are only to be used by the GridServer and are included by default. The important functions are discussed in section 3.3.7.
File: $GRIDROOT/libs/exec.pl†
Perl:include "$PATH{SYSTEMLIB}/exec.pl";

GridServer-Slots

The slot serves as a temporary storage container for all kinds of data and has to be created and destroyed explicitly, therefore its lifetime may exceed the scope of $\ulcorner JOBS \urcorner$ and **{COMMANDS}**. The library is not included by default. The explanation of some functions below will refer to the columns in table Tab.2.3. The column-names refer to table2.3.
File: $GRIDSRC/libs_perl/gridsrv/srvutils.pl†
Perl: include"$PATH{SRVLIB}/srvutils.pl";

- **&srv__reserveslot(*$workdir*)**: Creates a new slot, which automatically expires after 10 minutes, if it is not used and updated in the meantime. The optionally supplied $workdir will be written into the corresponding column **(workdir)**, otherwise the same temporary directory stored in column **(tempdir)** will be used instead. Usually an RPC-command other than **{slot.reserve}**$_S$ does not have to call this function explicitly, since a larger workflow should be composed like illustrated in Fig.2.20. In such a case the GridClient will create the slot by the aid of the command **{slot.reserve}**$_S$ in advance. The status of the slot is by default '0' (INACTIVE).
- **&srv__releaseslot(\\@*slots*)**: Removes the slot(s) from the registry and deletes the temporary directory **(tempdir)**, whereas **(workdir)** -if different- is not removed. The command **{slot.release}**$_S$ uses this function.
- **&srv__slot_extend(*$slot_id,$time*)**: Extends the expiry-date **(expiration)** of the slot to the given value, which is **now plus $time**. The string may be composed of days, hours, minutes and seconds e.g "12d3h5m12s".
- **&srv__slot_seterror(*$slot_id*)**: Changes the **(status)** to '-1'.
- **&srv__slot_setactive(*$slot_id*)**: Changes the **(status)** to '1'.
- **&srv__slot_setfinished(*$slot_id*)**: Changes the **(status)** to '2'.

Apart from the provided functions, it is possible to manipulate the slot-content directly by the use of the wiensql-functions (see Fig.A.1).

```
my $slot_id=10;
my $SQL="select * from slots.server where slot_id=$slot_id";
my %DATA=&db_cmd_array($SQL);
if($DATA{COUNT}==0)
{
    &write_error("The slot ’$slot_id’ does not exist");
}
else
{
    &write_verbose("The original content is ’$DATA{parameter}’");
    my $set="parameter=’<A>test</A>’";
    &write_verbose("changing parameter to ’<A>test</A>’");
    my $SQL="update slots.server set $set where slot_id=$slot_id";
    if(!&db_cmd($SQL))
    {
            &write_error("could not update data");
    }
    else
    {
            &write_result("slot successfully updated");
    }
}
```

Figure A.1: Sample code for a manual modifi cation of the slot-registry by the aid of the database-functions

GridClient-Slots

The following functions, which must be included explicitly behave similar to the previous ones. The column-names refer to table2.4.

File: $GRIDSRC/libs_perl/gridclient/clientutils.pl[†]

Perl: include “$PATH{CLIENTLIB}/clientutils.pl”;

- **&client_reserveslot(*$program,$servername,$time,$dir*)**: Register a new slot with default-status ’0’ and returns the slot_id The $time is the expiry-time of the slot supplied as **now + $time** (see above). The name of the $program, the $servername and the directory $dir are optional and may be updated later.
- **&client_releaseslot(*$slot_id*)**: Removes the slot. In contrast to the related function of the corresponding GridServer library, the directory **(dir)** will not be removed!
- **&client_slot_extend(*$slot_id,$time*)**: extends the expiry of the slot to **now + $time** (as before).
- **&client_slot_seterror(*$slot_id*)**: Changes the **(status)** to ’-1’.
- **&client_slot_setactive(*$slot_id*)**: Changes the **(status)** to ’1’.
- **&client_slot_setfinished(*$slot_id*)**: Changes the **(status)** to ’2’.
- **&client_slot_setjob(*$slot_id,$job_id*)**: Links the slot with a $job_id.

- **&client_slot_setname(*$slot_id,$jobname*)**: Changes the value of column **(name)** (e.g. the CASENAME of a **`WIEN2k`**-calculation).
- **&client_slot_setserver(*$slot_id,$servername*)**: The name of the GridServer, where the calculation shall be submitted to.

The sample code illustrated in Fig.A.1 may be used also for the GridClient-slots, if the table-name in line 2 and line 13 is changed from 'slots.server' to 'slots.client'.

GridServer-Jobs (identical to GridClient-jobs)

Only the function &jobutils__newjob() is supposed to be used in **{COMMANDS}**, whereas all other functions are reserved for $\ulcorner\!\llcorner JOBS \lrcorner\!\urcorner$. The data-exchange of the latter kind of functions uses an internal buffer, which fetches and stores all data. Also the instructions to modify data are not transacted immediately, instead only the values in the buffer are modified. Similar to the logfiles, the transaction is performed upon the call of a single function: &jobutils__update(). If this call is omitted (for any reason), the modifications made in the scope of the $\ulcorner\!\llcorner JOB \lrcorner\!\urcorner$ are lost. By default, these functions are available to every $\ulcorner\!\llcorner JOB \lrcorner\!\urcorner$-script, whereas the respective library has to be included in **{COMMAND}**-scripts. The column-names refer to the table 2.2.
GridClient-File:$GRIDSRC/libs_perl/gridclient/jobutils.pl†
GridClient-Perl:include "$PATH{CLIENTLIB}/jobutils.pl";
GridServer-File:$GRIDSRC/libs_perl/gridsrv/jobutils.pl†
GridServer-Perl:include "$PATH{SRVLIB}/jobutils.pl";

- **&jobutils__newjob(*$job,$time,$param,$files,$slot_id*)**: Register a new $\ulcorner\!\llcorner \$job \lrcorner\!\urcorner_{S,C}$, which is scheduled for execution at the date **now +$time** (string formatted as before) and returns the **(job_id)**. The list of parameters $param is usually an XML-datagram, but this is up to the developer. The optional filelist $files is the datagram-style expression of the respective hash as obtained from &filelist__data2str() (see 6).
- **&jobutils__setnextexec(*$time*)**: Changes the next execution **(job_date)** of the job.
- **&jobutils__killjob()**: Removes the job from the registry.
- **&jobutils__getparameter()**: Returns a datagram-hash.
- **&jobutils__setparameter(\*%param*)**: Updates the column **(parameter)** of the $\ulcorner\!\llcorner JOB \lrcorner\!\urcorner$. The list must be passed as a reference to the datagram-hash, which is the default-method to use the parameter-column of the job-registry. If it is intended to store an unformatted string, the corresponding database-functions have to be used.

- **&jobutils__getslotdata()**: Returns a hash, which contains the column-names of the slot-registry. The data can only be fetched, if the **(slot_id)** has been specified, otherwise it will be empty.
- **&jobutils__getfiles()**: returns a file-list hash as read from column **(files)**.
- **&jobutils__update_files(\*%files*)**: Changes the file-list hash to \\%files.
- **&jobutils__update()**: Realises all instructed updates to the table or removes the $\ulcorner JOB \urcorner$.

Misc. GridClient-functions

This is a collection of some useful functions, column-names refer to the host-registry.
File: $GRIDSRC/libs_perl/gridclient/clientutils.pl†
Perl: include "$PATH{CLIENTLIB}/clientutils.pl";

- **&gridclient__searchhosts(*$string*)**: Accepts an argument-string, which may look like 'mem>1000,cpu>=3,prog=WIEN' and returns the wiensql-query result (HASH) of &db_cmd_rows() (see section 6).
- **&gridclient__findhost(*$host*)**: The string $host may either be the **(host_id)**, the **(host_ip)** or the **(host_name)**. The function returns the wiensql-query result of &db_cmd_array() (see section 6). The COUNT-element of the result-hash (e.g. $DATA{COUNT}) contains the number of matches. If it is less than 1, the host could not be found, if it is greater than 1 the input is ambiguous. The other elements of the hash are the column-names of the host-registry (to be used as e.g. $DATA{host_name}).
- **&gridclient__fork(*$parameter*)**: Forks a process and returns the new PID of the child. Usually the parent will retain the control of the logfile and the open connection to the client. By default no wiensql-connection will be established. To change this behaviour, an optional list of parameters $parameter may be supplied, which can look like this: 'wiensql=yes gridlog=child connection=child'[5].
- **&gridclient__execjob(*$job,$job_id*)**: Immediately runs a $\ulcorner \$job \urcorner_{S,C}$ in the background. the job must already be registered, hence it is mandatory to pass a valid $job_id in order to retrieve and manipulate the data. Since this function is supposed to be used from within a **{COMMAND}**, it is recommended to fork the process in advance, otherwise the $\ulcorner JOB \urcorner$ will run in the foreground and block the **{COMMAND}**.

[5]The illustrated string will create a new wiensql-connection for the child and gives the control of the open logfi le and also the control of the client connection to the child.

Misc. GridServer-functions

The functions are already included and serve a similar purpose like those for the GridClient.

- **&gridsrv__fork(*$parameter*)**: Same as &gridclient__fork().
- **&gridsrv__execjob(*$job,$job_id*)**: Same as &gridclient__execjob().

GridServer connection

The respective library file is already included in **{COMMAND}**- and $\lceil JOB \rceil$-scripts. It contains the client-functions for connecting to and interacting with a GridServer. A sample code is provided with Fig.A.8 on page 180, the line numbers will refer to it.
File: $GRIDSRC/libs_perl/gridsrv.pl†
Perl: include “$PATH{LIB}/gridsrv.pl”;

- **&gridsrv__connect(*$host*)**: Establishes a connection to a GridServer, which has to be in the host-registry. It is retrieved from the host-registry by the aid of the supplied string $host, which may either be the **(host_id)**, the **(host_ip)** or the **(host_name)** (see &gridclient__findhost()). The function returns ’1’ upon success or ’0’ in the case of an error (line 6)
- **&gridsrv__exec(*$command,$parameter*)**: Submits the **{$command}**$_S$ together with some arguments ’$parameter’ to the GridServer and returns three strings: result,error,warning (line 12).
- **&gridsrv__disconnect()**: Terminates the connection (line 22).

These three are the only functions, developers will need from this library, whereas others like &gridsrv__put() and &gridsrv__get() should not be used. Instead it is recommended to use &utils_shared__filetransfer() (see page 175).

Files

Files (or more accurately: filenames) are treated by **W2GRID** in an object-like manner. A hash, referred to as the %filelist, contains individual 'objects' used like that: %filelist{0...COUNT-1}. Each of them is another hash, which contains items like the NAME, the PATH and SIZE of a file. The predominant purpose of the filelist is to store and handle a large number of files by the use of a single variable. The entries may be turned easily into an XML-string in order to store the contents in the database and also reverted to the object again. For filetransfer it is sufficient to supply this object to the function &utils_shared__filetransfer() (see page 175). The capabilities are available by default.
File: $GRIDSRC/libs_perl/filelist.pl†
Perl: include "$PATH{LIB}/filelist.pl";

- **&new_filelist(*$local,$remote*)**: Generates a new filelist object and returns the respective hash %filelist. Both arguments are optional. The string $local is attached to the local path of the filenames, and $remote to the remote ones.

- **&filelist__additem(\*%filelist,$name,$location,$checkpoint,$parameter*)**: Adds a new item to the object $filelist. The $name of the file is an absolute path (and may also contain the asterisk '*' character). The $location is a string (either 'local' or 'remote') and indicates whether the file has to be copied from the client to the server or the other way round. Checkpoints allow to mark files with a certain label (e.g. 'error','finished','check','input') and to apply the function &utils_shared__filetransfer() only to a selected sublist, which bears this label. An additional $parameter can be used to influence the transfer-method. By default the file is copied as a whole. If the string 'append' is supplied, only the most recent chunk will be appended.

- **&filelist__data2str(\*%filelist*)**: The filelist cannot be stored directly in the database, because it is represented by an internal Perl-address. For this purpose, it has to be turned into an XML-string. The result of this function may be stored directly in any sufficiently spacious text-type column.

- **&filelist__str2data(*$string*)**: Reverts the function &filelist__data2str(\*%filelist*) in order to get back the %filelist out of the XML-string.

- **&filelist__mklist(\*%filelist*)**: Concats all items contained in the list into a single string, which may be used e.g. as an argument for a command like ***tar***.

- **&filelist__compress(\*%filelist,$archive*)**: Compresses the files and returns either '1' upon success or '0' on error. The archive name $archive must be supplied as an absolute path, its extension will define the type of the archive and the applied method (at present only .gz, .tgz, .tar.gz and .tar are valid).

- **&filelist__refresh(\*%filelist*)**: Walks through the items and collects new informations (SIZE, timestamps,...).

XML-datagrams

XML-like structures are frequently used in **W2GRID** for data transfer and to wrap complex parameters or results. A sample code is shown in Fig.A.10 (page 181) that illustrates their use.
File: $GRIDSRC/libs_perl/datagramm.pl†[6]
Perl: include “$PATH{LIB}/datagramm.pl”;

- **&new_dtgr(*$name,$param,$value*)**: Generates a new datagram (line 1) and returns it as a %HASH. The $name is the root-tag, which contains all other items[7]. Optionally some parameters $param and a $value may be specified[8]. Additional tags should not be passed by $value but instead added by using &dtgr__additem().
- **&dtgr__additem(\*%HASH,$name,$param,$value*)**: Adds new items (lines 2 - 6). The function returns a reference to the recently created object, which may in turn be used to add new tags to it (lines 5 - 6).
- **&dtgr__readitem(\*%HASH,$name*)**: Reads the value of items (lines 9 - 11).
- **&dtgr__readparam(\*%HASH,$name,$element*)**: Reads the parameter of items (line 12)
- **&dtgr__countmembers(\*%HASH,$name*)**: Counts the number of items having the same name (e.g. <DATA><A></A><A></A></DATA>, <A> occurs two times;line 13)
- **&dtgr__countitems(\*%HASH,$name*)**: Counts the number of items in total, which have been added to a certain tag. (e.g. <DATA><A></A><B></B></DATA>, contains two items; line 14)
- **&dtgr__data2str(\*%HASH*)**: Turns the hash into a string in order to store it in e.g. a database-table. The string is returned (line 7).

[6]The original name has been kept, to avoid conflicts, although it is misspelled.
[7]e.g. <DATA>...</DATA>
[8]e.g. <DATA $param>$value</DATA>

- **&str2dtgr(*$string*)**: Processes a string and returns the datagram. This is useful to turn database-results from strings back into objects (HASH) (line 8).

wiensql

A very important feature is the database-connectivity, which is available to all **{COMMANDS}** and ⌜*JOBS*⌟ by default. In many cases it can be avoided to use the illustrated functions, because some regularly requested tasks are already provided with appropriate template-functions, which contain the SQL query as part of their workflow and reduce the amount of code-lines (e.g. manipulating slots and ⌜*JOBS*⌟). A sample script, which illustrates the use of the functions is shown in Fig.A.1. The library is already included by default.
File: $GRIDSRC/libs_perl/wiensql.pl†
Perl: include "$PATH{LIB}/wiensql.pl";

- **&wiensql__connect()**: Connects to the database and returns either '1' upon success or '0' on error.
- **&wiensql__disconnect()**: Terminates the connection.
- **&db_cmd(*$SQL*)**: Executs an SQL statement and returns the result. This function is useful for deleting or updating data (line 14), but not recommended for 'SELECT' statements, since the recordset will be obtained as an unformatted string (line 14).
- **&db_cmd_array(*$SQL*)**: Returns only the first row of a query, assuming, the query leads to a single match. The data is returned as hash, its elements are named after the column-names (e.g. 'parameter' line 10). An additional entry COUNT (line 4) indicates how many entries matched the query in total, thus this allows to determine whether the selection criteria leads to ambiguous results.
- **&db_cmd_rows(*$SQL*)**: Returns a complete recordset as a hash. The element COUNT contains the total number of rows. Each individual row is stored as an item bearing the respective index: 0...$COUNT-1 (e.g. $result{0} contains the first row). Each row is again a hash, containing elements, which are named after the columns (see Fig.A.2).

```
my %DATA=&db_cmd_rows("select * from slots.server");
for(my $i=0;$i<$DATA{COUNT};$i++)
{
    print "$DATA{$i}{slot_id}\n";
}
```

Figure A.2: Sample code of &db_cmd_rows()

Input

Arguments may be supplied to tools and commands[9]. Hence it is necessary to process these strings and extract the desired informations. The respective functions are already provided by default. A sample code is shown in Fig.A.4 (page 177), the line numbers in this section refer to it.
File: $GRIDSRC/libs_perl/input.pl†
Perl: include "$PATH{LIB}/input.pl";

- **&input__split(*$string*)**: Splits the arguments (supplied as $string) into individual pieces and stores them in an intermediate buffer. The function does not return a result.

- **&input__argument(*$param,$name,$regexp,$errmsg*)**: Checks if the argument $name (e.g. '--help') exists in the input string. If it is supposed to be a flag, $regexp and $errmsg must be empty, otherwise it is regarded to be an option, which requires an additional value to be attached to $name. A flag and an option must not have the same $name. The string $regexp may be any regular expression or a template such as ('NUMBER', 'FILE', 'DIR', 'INTEGER' or 'STRING'). The $errmsg will be returned (see function &write_error()) if either the option does not exist at all in the input-string, if no value has been passed or if the value does not match the given $regexp. $param may contain an additional definition of the argument (e.g. 'mandatory'). The function returns either '1' in the case of success or '0' on error.

- **&input__arguments(*$param,$names,$regexp,$errmsg*)**: Behaves identically to &input__argument(), but it accepts several $names, which must be separated by a slash (e.g. -h/--help) and allows to process many arguments at once.

- **&input__freeparam(*$param,$regexp,$errmsg*)**: Some parameters are neither options nor flags (e.g. just a number or a filename supplied to a **{COMMAND}** or tool). The behavior is identical to &input__argument(), yet the $name is missing. $regexp is mandatory.

- **&input__read()**: Every time an option or free parameter is successfully retrieved from the string, its value will be stored internally and may be retrieved by the aid of this function.

- **&input__finish()**: Checks if some arguments are still in the stack, which have not been asked for. All such items (e.g. misspelled ones) are considered to be inadequate and

[9]⌜*JOBS*⌟ in the contrary use other methods to retrieve their input (see above).

reported as an error. It is therefore recommended to check for errors before the program/command enters the main-part (line 24).

Shared utilities

The shared utilities are very important in **W2GRID**, since they offer the most useful capabilities for client/server command interplay and complex processes. They have to be included on demand.
File: $GRIDSRC/libs_perl/gridshared/utils_shared.pl†
Perl: include “$PATH{SHAREDLIB}/utils_shared.pl”;

- **&utils_shared__getcalcstate(*$slot_id,$job_id*)**: The function can only be used, if a connection to a GridServer has already been established successfully (&gridsrv__connect()). It returns the state of a remote calculation, which is a combination of the states of the ⌜*JOB*⌟ and the slot. The return value is a string out of the following list: ’FATAL’ ’ERROR’ ’CRAZY’ ’NOSTART’ ’CRASHED’ ’RUNNING’ ’FINISHED’. ’FATAL’ is returned if any of the two states could not be obtained. A state is CRAZY if the slot-state and the ⌜*JOB*⌟-state do not match (e.g. the ⌜*JOB*⌟ exists but the slot is already finished). A calculation is ’CRASHED’, if the slot is recognised to be still active, but the controlling ⌜*JOB*⌟ is gone. This is different from ERROR.

- **&utils_shared__filetransfer(\*%filelist,$slot_id,$param*)**: This function needs an open GridServer connection, too. The filelist object %filelist and the remote $slot_id are mandatory arguments. The transfers is performed in both directions, according to the given $location of a file (see function &filelist__additem on page 171). A ’remote’ file will be copied from the server to the client, whereas a ’local’ file will be transferred the other way round. Additional constraints $param allow to pre-select the files (e.g. ’checkpoint=finished’ only transfers files, which are labelled as ’finished’).

- **&utils_shared__parallel_processes(\*@input,$function*)**: A sample code is shown in Fig.A.6 (page 179). The function invokes asynchronous parallel processes, which receive exactly one element out of the array of input data. The number of invoked parallel processes equals the number of elements contained in the array. The string $function contains the name of the function, which shall be executed by each process. The results of each single function will be collected and returned as an array. If the number of elements of the output-array does not match the number of elements of the input, an error has occurred.

Sample Code and Screenshots

All sample code shown in this section will use line numbers to simplify referring to certain functionalities from other parts of the text. If using these scripts for development, the line number must of course not be included.

```
 1: #!/bin/csh
 2: # @ job_type = parallel
 3: # @ input = /dev/null
 4: # @ output = $(Executable).$(Cluster).$(Process).out
 5: # @ error = $(Executable).$(Cluster).$(Process).err
 6: # @ notify_user = pblaha@susi.theochem.tuwien.ac.at
 7: # @ node_usage = not_shared
 8: # @ initialdir = ~/lapw/LiF
 9: # @ tasks_per_node = 2
10: # @ node = 8
11: # @ class = large
12: # @ queue
13: set mpijob=1
14: setenv OMP_NUM_THREADS 1
15: limit coredumpsize 0
16: set proclist=‘echo $LOADL_PROCESSOR_LIST‘
17: set nproc=$#proclist
18: echo ’#’ > .machines
19: #k-point and mpi parallel lapw1/2
20: set i=1
21: while ($i <= $nproc )
22: echo -n ’1:’ >>.machines
23: @ i1 = $i + $mpijob
24: @ i2 = $i1 - 1
25: echo $proclist[$i-$i2] >>.machines
26: set i=$i1
27: end
28: echo ’granularity:1’ >>.machines
29: echo ’extrafine:1’ >>.machines
```

Figure A.3: submission-script for a **`WIEN2k`** calculation on the LoadLeveller (LL)

```
#Creation:20.2.04
#Author:Johannes Schweifer
#+-------------------------------------------------+
#+               REQUIRE LIBRARIES                 +
#+-------------------------------------------------+

require "$PATH{SRVLIB}/srvutils.pl";

#+-------------------------------------------------+
#+                MAIN ROUTINES                    +
#+-------------------------------------------------+
sub exec_request
{

    #----------------------------------------
    # Evaluating commandline arguments
    #----------------------------------------

    if(&input__arguments("","-h/--help"))
    {
            &write_result(&test__help());
            &set_finished();
    }
    &input__finish();

    #----------------------------------------
    # Main code
    #----------------------------------------

    if(!&check_abort())
    {
            &write_result("If you read this the gridserver is operational");
    }

    #----------------------------------------
    # Send answer ...
    #----------------------------------------

    return &gridsrv__make_response();
}

sub exec_help
{
    #----------------------------------------
    # Available options and flags (without explanation!)
    #----------------------------------------
    &write_result(&w2grid__getcommand()." [-h/--h]");
    return &gridsrv__make_response();
}

#+-------------------------------------------------+
#+                      HELP                       +
#+-------------------------------------------------+

sub test__help
{
    my $commandname=&w2grid__getcommand();
    my $output=<<_END;

USAGE:            $commandname [-flags]

PURPOSE:          On the one hand a template for further
                  commands. On the other hand a test for
                  the server.
FLAGS:
-h/--help     --> displays this text

_END
return $output;
}
1;
```

Figure A.4: Source code of the **W2GRID** version of 'Hello World' **{test}**$_S$

```
#Creation:17.11.04
#Author:Johannes Schweifer
#+-------------------------------------------------+
#+               REQUIRE LIBRARIES                 +
#+-------------------------------------------------+

my $global_variable=10;

#+-------------------------------------------------+
#+                MAIN ROUTINES                    +
#+-------------------------------------------------+

sub exec_command
{
    #----------------------------------------
    # Evaluating stored parameters
    #----------------------------------------

    my %jobdtgr=%{&jobutils__getdata("PARAMETER")};
    my %slotdata=&jobutils__getslotdata();

    #----------------------------------------
    # Main code
    #----------------------------------------

    if(&jobutils__isstate("init"))
    {
            &write_verbose("The job is in the initialization phase");
            &run_external("echo initialization' >> $PATH{LOG}/jobtest.txt");
            &jobutils__changestate("running");
            &write_verbose("Next execution in now+90 seconds");
            &jobutils__setnextexec("90s");
    }
    else
    {
            &run_external("echo done' >> $PATH{LOG}/jobtest.txt");
            &write_verbose("The job will not be executed again");
            &jobutils__killjob();
    }

    #----------------------------------------
    # Clean up and handle next execution
    #----------------------------------------

    &jobutils__setparameter(\%jobdtgr);
    &jobutils__update();
 }
 1;
```

Figure A.5: Source code of a background task ('job' **{test}**$_S$)

```
#+--------------------------------------------------+
#+               REQUIRE LIBRARIES                  +
#+--------------------------------------------------+
require "$PATH{SHAREDLIB}/utils_shared.pl";
#+--------------------------------------------------+
#+                MAIN ROUTINES                     +
#+--------------------------------------------------+
sub exec_request
{
    #----------------------------------------
    # Evaluating commandline arguments
    #----------------------------------------
    if(&input__arguments("","-h/--help"))
    {
        &write_result(&test__help());
        &set_finished();
    }
    &input__finish();
    #----------------------------------------
    # Main code
    #----------------------------------------
    if(!&check_abort())
    {
            #first we create some input
            my $result="";
            my @input=(1,2,3,4,5,6,7,8,9,10);
            my @output=&utils_shared__parallel_processes(\@input,\&pfunc,100);
            foreach(@output)
            {
                $result.="$_\n";
            }
            $result.="10 jobs were launched. x^^2 with x in [2-11]";
            &write_result($result);
    }
    #----------------------------------------
    # Send answer ...
    #----------------------------------------
    return &gridclient__make_response();
}
sub exec_help
{
    #----------------------------------------
    # Available options and flags (without explanation!)
    #----------------------------------------
    &write_result(&w2grid__getcommand()." [-h/--h]");
    return &gridclient__make_response();
}
sub pfunc
{
    #here we receive some input
    #the input is used for a calculation...
    my $input=$_[0];
    sleep(1);
    &demon__progress(50);
    sleep(1);
    &demon__progress(50);
    return ($input+1)*($input+1);
}
#+--------------------------------------------------+
#+                     HELP                         +
#+--------------------------------------------------+
sub test__help
{
    my $commandname=&w2grid__getcommand();
    my $output=<<_END;
USAGE:             $commandname [-flags]
PURPOSE:           shows parallel processes
FLAGS:
-h/--help      --> displays this text
_END
return $output;
}
1;
```

Figure A.6: Parallel sample command-code (GridClient and GridServer compliant)

```
my $templates="$PATH{TEMP}/template=>$PATH{TEMP}/pbsnodelist";
my $command="sleep 100";
my %parameter=(
               "MPI"  =>1,
               "CPU"  =>10,
               "MEM"  =>200, #MB
               "TIME" =>10h, #time-string
               "PREAMBLE" =>"", #some commands
               "NOTIFY" =>"me@home",
               "FILES"=>$templates,
               "TASKS_PER_NODE"=>2,
               "THREADS_PER_NODE"=>2,
               "ERRORFILE"=>"test.err",
               "OUTPUTFILE"=>"test.out",
               "INPUTFILE"=>"test.in",
               "COPYINPUT"=>1,    #MPI:to each node
               "COPYPROGRAM"=>1,  #MPI:to each node
               "COPYOUTPUT"=>1,   #MPI:from each node
               );
$ID=&exec__submit($PATH{TEMP},$command,\%parameter);
```

Figure A.7: The using of plugin-functions in order to submit a job to a queuing system

```
my %HOST=(
              "host_name"=>"athena",
              "host_id"=>"12",
              "slot_id"=>23,
              );
if(!&gridsrv__connect("$HOST{host_id}"))
{
    &write_error("could not connect to '$HOST{host_name}'");
}
else
{
    my ($data,$error,$warning)=&gridsrv__exec("slot.release","$HOST{slot_id}");
    if($error!~/\w/)
    {
            &write_verbose("The remote slot $HOST{slot_id} has been released");
            return 1;
    }
    else
    {
            &write_error($error);
    }
    &gridsrv__disconnect();
}
```

Figure A.8: Sample code for sending a remote-procedure-call to a GridServer

```
my %jobdtgr=%{&jobutils__getparameter()};
my $local_slot_id=&dtgr__readitem(\%jobdtgr,"PARAMETER.LOCAL_SLOT_ID");
my $str_machines=&dtgr__readitem(\%jobdtgr,"PARAMETER.MACHINES");
my $maxnodes=&dtgr__readitem(\%jobdtgr,"PARAMETER.MAXNODES");
my $processes=&dtgr__readitem(\%jobdtgr,"PARAMETER.PROCESSES");
my $maxmem=&dtgr__readitem(\%jobdtgr,"PARAMETER.MAXMEM");
my $program=&dtgr__readitem(\%jobdtgr,"PARAMETER.PROGRAM");
my $str_wienvar=&dtgr__readitem(\%jobdtgr,"PARAMETER.WIENVAR");
my $logfile=&dtgr__readitem(\%jobdtgr,"PARAMETER.LOGFILE");
my $keep=&dtgr__readitem(\%jobdtgr,"PARAMETER.KEEP");
my $cycle=&dtgr__readitem(\%jobdtgr,"PARAMETER.CYCLE");
my $local_pid=&dtgr__readitem(\%jobdtgr,"PARAMETER.PID");
my $outfile=&dtgr__readitem(\%jobdtgr,"PARAMETER.OUTFILE");
my $errfile=&dtgr__readitem(\%jobdtgr,"PARAMETER.ERRFILE");
my $wien2kparam=&dtgr__readitem(\%jobdtgr,"PARAMETER.WIEN2KPARAM");
```

Figure A.9: Sample code for retrieving data from the parameter-column of the job-registry

```
my %dtgr=&new_dtgr("DATA");  #<DATA></DATA>
&dtgr__additem(\%dtgr,"A","",1); #<DATA><A>1</A></DATA>
&dtgr__additem(\%dtgr,"A","",10); #<DATA><A>1</A><A>10</A></DATA>
&dtgr__additem(\%dtgr,"B","",2); #<DATA>...<B>2</B></DATA>
my $c=&dtgr__additem(\%dtgr,"C","test=yes",3); #returns reference
&dtgr__additem($c,"X","","hello"); #<DATA>...<C test=yes><X>hello</X></C>...
my $string=&dtgr__data2str(\%dtgr); #returns the string
my %new_dtgr=&str2dtgr($string); #returns the hash
my $a1=&dtgr__readitem(\%dtgr,"DATA.A[0]"); #returns 1
my $a2=&dtgr__readitem(\%dtgr,"DATA.A[1]"); #returns 10
my $x=&dtgr__readitem(\%dtgr,"DATA.C.X"); #returns 10
my $param=&dtgr__readparam(\%dtgr,"DATA.C.X","test"); #returns yes
my $count_a=&dtgr__countmembers(\%dtgr,"DATA.A"); #returns 2
my $items=&dtgr__countitems(\%dtgr,"DATA"); #returns 3 (items:A,B,C)
```

Figure A.10: Sample code illustrating the manipulation of XML-datagrams

```
<PLUGIN>
    <ID>_PLAT_REDHAT</ID>
    <TYPE>PLATFORM</TYPE>
    <NAME>RedHat</NAME>
    <CUSTOM>
        <ITEM>
            <NAME>platform</NAME>
            <REGEXP empty=true default=unknown>STRING</REGEXP>
            <DESCRIPTION>The redHat-linux version</DESCRIPTION>
            <QUERY>Supply the version number if you know it:</QUERY>
        </ITEM>
    </CUSTOM>
    <FIXED>
        <ITEM>
            <NAME>system_file</NAME>
            <VALUE>libs_perl/platforms/redhat.pl</VALUE>
        </ITEM>
        <ITEM>
            <NAME>endian</NAME>
            <VALUE>0</VALUE>
        </ITEM>
    </FIXED>
    <DESCRIPTION>Special for RedHat-versions</DESCRIPTION>
</PLUGIN>
```

Figure A.11: Sample XML-descriptor of a platform plugin

```
<PLUGIN>
    <ID>_CPU_P4_3.2</ID>
    <TYPE>CPU</TYPE>
    <NAME>Pentium 4 (3.2 GHz)</NAME>
    <CUSTOM></CUSTOM>
    <FIXED>
        <ITEM>
            <NAME>SPEED</NAME>
            <VALUE>650</VALUE>
        </ITEM>
    </FIXED>
    <DESCRIPTION></DESCRIPTION>
</PLUGIN>
```

Figure A.12: Sample XML-descriptor of a processor plugin

```
<PLUGIN>
    <ID>_FTP_SCP</ID>
    <TYPE>FTP</TYPE>
    <NAME>unix secure-copy (scp)</NAME>
    <CUSTOM>
        <ITEM>
            <NAME>protocol</NAME>
            <REGEXP default=2>^(1|2)$</REGEXP>
            <DESCRIPTION>The version of the RSA algorithm</DESCRIPTION>
            <QUERY>Please enter the RSA version (default=2):</QUERY>
        </ITEM>
        <ITEM>
            <NAME>ip</NAME>
            <REGEXP>IP</REGEXP>
            <DESCRIPTION>scp xy.txt IP:(path)</DESCRIPTION>
            <QUERY>Enter the IP/hostname of the remote host:</QUERY>
        </ITEM>
        <ITEM>
            <NAME>login</NAME>
            <REGEXP>STRING</REGEXP>
            <DESCRIPTION>for ssh login@host</DESCRIPTION>
            <QUERY>Enter the login name on the remote host:</QUERY>
        </ITEM>
    </CUSTOM>
    <FIXED>
        <ITEM>
            <NAME>ftp_init</NAME>
            <DESCRIPTION>Initializing the FTP-connection</DESCRIPTION>
            <VALUE>standard_ftp_init</VALUE>
        </ITEM>
        <ITEM>
            <NAME>ftp_get</NAME>
            <DESCRIPTION>Copy data from remote to local</DESCRIPTION>
            <VALUE>standard_scp_get</VALUE>
        </ITEM>
        <ITEM>
            <NAME>ftp_put</NAME>
            <DESCRIPTION>Copy data from local to remote</DESCRIPTION>
            <VALUE>standard_scp_put</VALUE>
        </ITEM>
        <ITEM>
            <NAME>file</NAME>
            <DESCRIPTION>Relative path to the library-file</DESCRIPTION>
            <VALUE>ftps/rscp.pl</VALUE>
        </ITEM>
    </FIXED>
    <DESCRIPTION>scp is currently the safest data transfer method. Use it
    as a default method on any kind of system, where you have an account.
    The automatic authentication must be enabled (ssh-keygen!). If you still
    have to provide a password on login, the method will fail</DESCRIPTION>
</PLUGIN>
```

Figure A.13: Sample XML-descriptor of a fi letransfer plugin

```xml
<PLUGIN>
    <ID>_CONN_SOCKET</ID>
    <TYPE>CONNECTION</TYPE>
    <NAME>socket</NAME>
    <CUSTOM>
        <ITEM>
            <NAME>ip</NAME>
            <REGEXP>IP</REGEXP>
            <DESCRIPTION>The remote IP</DESCRIPTION>
            <QUERY>Please enter the remote IP:</QUERY>
        </ITEM>
        <ITEM>
            <NAME>port</NAME>
            <REGEXP>INTEGER</REGEXP>
            <DESCRIPTION>The remote port</DESCRIPTION>
            <QUERY>Please enter the remote port:</QUERY>
        </ITEM>
    </CUSTOM>
    <FIXED>
        <ITEM>
            <NAME>connection_open</NAME>
            <DESCRIPTION>Open a remote connection</DESCRIPTION>
            <VALUE>standard_socket_open</VALUE>
        </ITEM>
        <ITEM>
            <NAME>connection_write</NAME>
            <DESCRIPTION>Routine for sending data</DESCRIPTION>
            <VALUE>standard_socket_write</VALUE>
        </ITEM>
        <ITEM>
            <NAME>connection_read</NAME>
            <DESCRIPTION>Routine for reading data</DESCRIPTION>
            <VALUE>standard_socket_read</VALUE>
        </ITEM>
        <ITEM>
            <NAME>connection_status</NAME>
            <DESCRIPTION>Checking the connection status</DESCRIPTION>
            <VALUE>standard_socket_status</VALUE>
        </ITEM>
        <ITEM>
            <NAME>connection_close</NAME>
            <DESCRIPTION>Closing a remote connection</DESCRIPTION>
            <VALUE>standard_socket_close</VALUE>
        </ITEM>
        <ITEM>
            <NAME>connection_init</NAME>
            <DESCRIPTION>Prepare for opening the connection</DESCRIPTION>
            <VALUE>standard_socket_init</VALUE>
        </ITEM>
        <ITEM>
            <NAME>file</NAME>
            <DESCRIPTION>Relative library-path</DESCRIPTION>
            <VALUE>connections/socket.pl</VALUE>
        </ITEM>
        <ITEM>
            <NAME>name</NAME>
            <DESCRIPTION>The internal socket-descriptor</DESCRIPTION>
            <VALUE>GRIDSRV</VALUE>
        </ITEM>
    </FIXED>
    <DESCRIPTION>Standard unix socket employs a direct connection from one
    endpoint(specified by ip:port) to the remote one. If remote ports are
    blocked you can use an ssh-tunneling mechanism</DESCRIPTION>
</PLUGIN>
```

Figure A.14: Sample XML-descriptor of a connection plugin

```
<PLUGIN>
    <ID>_EXEC_PBS</ID>
    <TYPE>EXEC</TYPE>
    <NAME>PBS</NAME>
    <CUSTOM>
        <ITEM>
            <NAME>nodes</NAME>
            <REGEXP empty=true default=1>INTEGER</REGEXP>
            <QUERY>Please enter the number of nodes:</QUERY>
        </ITEM>
        <ITEM>
            <NAME>maxnodes</NAME>
            <REGEXP empty=true>INTEGER</REGEXP>
            <QUERY>The maximum number of nodes for a single job:</QUERY>
        </ITEM>
        <ITEM>
            <NAME>cpus</NAME>
            <REGEXP empty=true default=1 set="gt(1)">INTEGER</REGEXP>
            <QUERY>Please enter the number of CPUs per node:</QUERY>
        </ITEM>
        <ITEM>
            <NAME>cores</NAME>
            <REGEXP empty=true default=1 set="gt(1)">INTEGER</REGEXP>
            <QUERY>Please enter the number of cores per CPU:</QUERY>
        </ITEM>
        <ITEM>
            <NAME>threadvar</NAME>
            <REGEXP empty=true condition="$cores||$cpus">STRING</REGEXP>
            <QUERY>Threading variable (e.g.:OMP_NUM_THREADS):</QUERY>
        </ITEM>
        <ITEM>
            <NAME>mem</NAME>
            <REGEXP>INTEGER</REGEXP>
            <QUERY>Enter the memory/node or total if shared:</QUERY>
        </ITEM>
        <ITEM>
            <NAME>shm</NAME>
            <REGEXP empty=true default=1>(yes|no|y|n|1|0)</REGEXP>
            <QUERY>Is this memory shared among all nodes? (y/n):</QUERY>
        </ITEM>
        <ITEM>
           <NAME>shell</NAME>
           <REGEXP>STRING</REGEXP>
           <QUERY>Shell (e.g.: /bin/bash, or /bin/csh):</QUERY>
        </ITEM>
        <ITEM>
            <NAME>mpi</NAME>
            <REGEXP empty=true default=0 set="reg(yes|y)">(yes|no|y|n)</REGEXP>
            <QUERY>Do you want to enable MPI? (y/n):</QUERY>
        </ITEM>
        <ITEM>
            <NAME>mpiinit</NAME>
            <REGEXP empty=true condition=$mpi>STRING</REGEXP>
            <QUERY>The command, which prepares your MPI-environment:</QUERY>
        </ITEM>
        <ITEM>
            <NAME>mpirun</NAME>
            <REGEXP empty=false condition=$mpi>STRING</REGEXP>
            <QUERY>The MPI-string (see variables in the usersguide):</QUERY>
        </ITEM>
        <ITEM>
            <NAME>mpicleanup</NAME>
            <REGEXP empty=true condition=$mpi>STRING</REGEXP>
            <QUERY>Enter a command, which cleans your MPI-environment:</QUERY>
        </ITEM>
    </CUSTOM>
    <FIXED>
        <ITEM>
            <NAME>system_file</NAME>
            <VALUE>libs_perl/execs/pbs.pl</VALUE>
        </ITEM>
    </FIXED>
</PLUGIN>
```

Figure A.15: Sample XML-descriptor of a execution plugin

```
:etc
<header>This will install the WIEN2k-plugin</header>
<footer>The installation is done, try ’wien.test’</footer>
:envvars
<ENVVAR>
    <NAME>WIENROOT</NAME>
    <MISSING>EXIT</MISSING>
    <ON_MISSING>Check your WIEN2k-installation!</ON_MISSING>
</ENVVAR>
<ENVVAR>
    <NAME>GRIDSRC</NAME>
    <MISSING>EXIT</MISSING>
    <ON_MISSING>Check your W2GRID-installation!</ON_MISSING>
</ENVVAR>
:variables
<VARIABLE>
    <NAME>WIENROOT</NAME>
    <QUERY>The path of your WIENROOT (default: $ENV{WIENROOT})</QUERY>
    <REGEXP default="$ENV{WIENROOT}";empty=true>PATHNAME</REGEXP>
    <ERROR>The path does not exist</ERROR>
</VARIABLE>
<VARIABLE>
    <NAME>HEURISTICS</NAME>
    <QUERY>Do you want to use heuristics [y/n]?\n(default: yes)</QUERY>
    <REGEXP default="1";empty=true>[01ynYN]</REGEXP>
    <ERROR>say y(es) or n(o)</ERROR>
</VARIABLE>
:directories_install
<DIRECTORY>
    <NAME>$VAR{WIENROOT}</NAME>
    <MISSING>EXIT</MISSING>
    <ON_MISSING>The WIENROOT-directory MUST exist!</ON_MISSING>
</DIRECTORY>
<DIRECTORY>
    <NAME>$ENV{GRIDSRC}/SRC_gridsrv/commands/wien</NAME>
    <MISSING>CREATE</MISSING>
    <ON_MISSING>$SRC_gridsrv/commands/wien will be created</ON_MISSING>
</DIRECTORY>
<DIRECTORY>
    <NAME>$ENV{GRIDSRC}/SRC_gridsrv/jobs/wien</NAME>
    <MISSING>CREATE</MISSING>
    <ON_MISSING>SRC_gridsrv/jobs/wien will be created</ON_MISSING>
</DIRECTORY>
<DIRECTORY>
    <NAME>$ENV{GRIDSRC}/libs_perl/wien</NAME>
    <MISSING>CREATE</MISSING>
    <ON_MISSING>libs_perl/wien will be created</ON_MISSING>
</DIRECTORY>
<DIRECTORY>
    <NAME>$ENV{GRIDSRC}/libs_perl/wien/gridsrv</NAME>
    <MISSING>CREATE</MISSING>
    <ON_MISSING>libs_perl/wien/gridsrv will be created</ON_MISSING>
</DIRECTORY>
:dbinserts
<DBINSERT>
    <TABLENAME>programs.server</TABLENAME>
    <EXIST query=true>UPDATE</EXIST>
    <ON_EXIST>update entry? (y/n) </ON_EXIST>
    <FAIL>WARN</FAIL>
    <ON_FAIL>The update failed.</ON_FAIL>
    <INSERT_ID>program_id</INSERT_ID>
    <options>
        <PARAMETER><HEURISTICS>$VAR{HEURISTICS}</HEURISTICS><WIENROOT>
        $VAR{WIENROOT}</WIENROOT><COUNT>1</COUNT></PARAMETER>
    </options>
    <program unique=true>WIEN</program>
    <speed>1</speed>
    <version unique=true>2k</version>
</DBINSERT>
```

Figure A.16: Sample installation instructions for the GridServer **`WIEN2k`**-package

```
:etc
<header>This will remove the GridServer WIEN2k-plugin</header>
<footer>Plugin removed.</footer>
:envvars
<ENVVAR>
    <NAME>GRIDSRC</NAME>
    <MISSING>EXIT</MISSING>
    <ON_MISSING>Check your W2GRID installation</ON_MISSING>
</ENVVAR>
:directories_uninstall
<DIRECTORY>
     <QUERY></QUERY>
     <NAME>$ENV{GRIDSRC}/SRC_gridsrv/commands/wien</NAME>
     <NOT_EXIST>WARN</NOT_EXIST>
     <ON_NOT_EXIST>commands/wien does not exist</ON_NOT_EXIST>
</DIRECTORY>
<DIRECTORY>
     <QUERY></QUERY>
     <NAME>$ENV{GRIDSRC}/SRC_gridsrv/jobs/wien</NAME>
     <NOT_EXIST>WARN</NOT_EXIST>
     <ON_NOT_EXIST>jobs/wien does not exist</ON_NOT_EXIST>
</DIRECTORY>
<DIRECTORY>
     <QUERY></QUERY>
     <NAME>$ENV{GRIDSRC}/libs_perl/wien/gridsrv</NAME>
     <NOT_EXIST>WARN</NOT_EXIST>
     <ON_NOT_EXIST>libs_perl/wien/gridsrv does not exist</ON_NOT_EXIST>
</DIRECTORY>
<DIRECTORY>
     <QUERY>Keep libs_perl/wien for the GridClient (y/n)?</QUERY>
     <NAME>libs_perl/wien</NAME>
     <NOT_EXIST>WARN</NOT_EXIST>
     <ON_NOT_EXIST>libs_perl/wien does not exist</ON_NOT_EXIST>
</DIRECTORY>
:files_uninstall
<FILE>
     <QUERY>Keep bin/memory_lapw.pl for GridClient (y/n)?</QUERY>
     <NAME>$ENV{GRIDSRC}/bin/memory_lapw.pl</NAME>
     <NOT_EXIST>WARN</NOT_EXIST>
     <ON_NOT_EXIST>bin/memory_lapw.pl does not exist!</ON_NOT_EXIST>
</FILE>
<FILE>
     <QUERY>Keep bin/calctime_lapw.pl for GridClient (y/n)?</QUERY>
     <NAME>$ENV{GRIDSRC}/bin/calctime_lapw.pl</NAME>
     <NOT_EXIST>WARN</NOT_EXIST>
     <ON_NOT_EXIST>bin/calctime_lapw.pl does not exist!</ON_NOT_EXIST>
</FILE>
<FILE>
     <QUERY>Keep bin/parameter_lapw.pl for GridClient (y/n)? </QUERY>
     <NAME>$ENV{GRIDSRC}/bin/parameter_lapw.pl</NAME>
     <NOT_EXIST>WARN</NOT_EXIST>
     <ON_NOT_EXIST>bin/parameter_lapw.pl does not exist!</ON_NOT_EXIST>
</FILE>
:dbdeletes
<DBDELETE>
    <TABLENAME>programs.server</TABLENAME>
    <QUERY></QUERY>
    <FAIL>WARN</FAIL>
    <ON_FAIL></ON_FAIL>
    <host_id>0</host_id>
    <program>WIEN</program>
    <version>2k</version>
</DBDELETE>
```

Figure A.17: Sample removal instructions for the GridServer **`WIEN2k`**-package

```
SRC_gridsrv/commands/wien/
SRC_gridsrv/jobs/wien/
libs_perl/wien/gridsrv
libs_perl/wien/in1.pl
libs_perl/wien/klist.pl
libs_perl/wien/machines.pl
libs_perl/wien/parameter.pl
libs_perl/wien/struct.pl
libs_perl/wien/wienfiles.pl
libs_perl/wien/wien.pl
libs_perl/wien/wienvar.pl
bin/memory_lapw.pl
bin/parameter_lapw.pl
bin/calctime_lapw.pl
```

Figure A.18: List of files, which will be included in the package (**`WIEN2k`** on the GridServer)

```
set hostname = ‘hostname‘
set oldlib = ""
if ($?PERLLIB) then
    set oldlib = "${PERLLIB}:"
else
    setenv PERLLIB ""
endif

if ("$hostname" == "linux") then
    setenv WIENSQL_ROOT /home/.wiensql_linux
    setenv PERLLIB ${oldlib}/home/.wgrid_linux/libs/i586-linux-thread-multi
    setenv GRIDROOT /home/.wgrid_linux
    setenv GRIDSRC /home/WGRID
    set path = ($path $GRIDROOT/bin $GRIDSRC/bin $GRIDSRC/SRC_wiensql/bin)
    set path = ($path $GRIDSRC/SRC_gridsrv/bin $GRIDSRC/SRC_gridclient/bin)
endif
```

Figure A.19: Sample code snippet of a .cshrc file (Modifications by **`W2GRID`**)

```
>host.info gescher
#command took 0 seconds to complete
HOSTNAME            gescher
ID                  4
IP                  131.130.186.180
PORT                8180
LAST-CHECK          15-11-2006
RELIABILITY         99.00 %
FAILED CONNECTIONS  6.49 %
RESPONSETIME        0 s
CPU                 1 processors / node
MEM                 4000 MB / node
SPEED               600 speed-units
NODES               16
CONNECTION          socket
                    Standard unix socket employs a direct connection from one
                    endpoint (specified by ip:port) to the remote one. If remote
                    ports are blocked you can use an ssh-tunneling mechanism
FTP                 unix secure-copy (scp)
                    scp is currently the safest data transfer method. Use it as a
                    default method on any kind of system, where you have an
                    account. The automatic authentication must be enabled
                    (ssh-keygen!). If you still have to provide a password on
                    login, the method will fail
PROGRAMS            WIEN
```

Figure A.20: Screenshot of the output of the command **{host.info}**$_C$

List of Figures

List of Tables

Bibliography

[1] H.J. Leisi. *Quantenphysik*. Springer-Verlag, 2006.

[2] W. Kutzelnigg. *Einführung in die Theoretische Chemie*. Wiley-VCH, 2002.

[3] Ch.J. Cramer. *Essentials of Computational Chemistry*. John Wiley & Sons, Ltd., 2004.

[4] *Atoms in Molecules, a quantum theory*. Oxford Science Publications, 1990.

[5] R.M. Dreizler and E.K.U. Gross. *Density Functional Theory*. Springer-Verlag, 1990.

[6] V. Staemmler. *Introduction to Hartree-Fock and CI Methods*, volume 31 of *NIC series*, chapter Computational Nanoscience: Do it Yourself. John von Neumann Institute for Computing, Jülich, 2006.

[7] W. Kohn and L.S. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev. B*, 140:1133, 1965.

[8] P. Blaha, K. Schwarz, and G.K.H. Madsen. Electronic structure calculations of solids using the WIEN2k package for materials sciences. *Comp. Phys. Commun.*, 147:71, 2002.

[9] P. Blaha, K. Schwarz, G.K.H. Madsen, D. Kvasnicka, and J. Luitz. *An Augmented Plane Wave Plus Local Orbital Program for Calculating Crystal Properties*. Number ISBN:3-9501031- 1-2 in WIEN2k. K.Schwarz, 2001.

[10] K. Schwarz. DFT calculations of solids with LAPW and WIEN2k. *Solid State Commun.*, 176:319, 2003.

[11] D. Laforenza. Grid programming: some indications where we are headed. *Parallel Comput.*, 28(12):1733–1752, 2002.

[12] K. Schwarz, P. Blaha, and J. Schweifer. From crystal structure to properties of solids with the grid-enabled WIEN2k. In Jens Volkert, Thomas Fahringer, Dieter Kranzlmueller,

and Wolfgang Schreiner, editors, *1st Austrian Grid Symposium*, pages 25–37, Schloss Hagenberg, Austria, December 2005. Austrian Computer Society.

[13] Photonics institute, vienna university of technology. http://info.tuwien.ac.at/photonik/index.htm. Gusshausstrasse 27/387, A-1040 Vienna.

[14] J. Zanghellini, M. Kitzler, C. Fabian, T. Brabec, and A. Scrinzi. An MCTDHF approach to multielectron dynamics in laser fields. *Laser Physics*, 13(8):1064–1068, 2003.

[15] J. Caillat, J. Zanghellini, M. Kitzler, O. Koch, W. Kreuzer, and A. Scrinzi. Correlated multielectron systems in strong laser fields: A multiconfiguration time-dependent hartree-fock approach. *Physical review A*, 71:012712, 2005.

[16] J. Caillat, J. Zanghellini, and A. Scrinzi. Parallelization of the MCTDHF code. Technical Report 19, AURORA, 2004.

[17] S.A. Jarvis, D.P.Spooner, H.N. Lim Choi Keung, J. Cao, S. Saini, and G.R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems*, 22:745–754, 2006.

[18] R.J. Allan and M. Ashworth. A survey of distributed computing, computational grid, meta-computing and network information tools. Daresbury, Warrington WA4 4AD, UK, 2001.

[19] A. Aizcorbe and S. Kortum. Moore's law and the semiconductor industry: A vintage model. Industrial Organization 0412008, EconWPA, Dec 2004. available at http://ideas.repec.org/p/wpa/wuwpio/0412008.html.

[20] K.A. Hawick, D.A. Grove, and F.A. Vaughan. Beowulf - A New Hope for Parallel Computing? In *Proc. of the 6th IDEA Workshop, Rutherglen*, 1999.

[21] Message Passing Interface Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):159–416, 1994.

[22] B. Mohr, J. Larsson Träff, J. Worringen, and J. Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Proceedings*, volume 4192 of *Lecture Notes in Computer Science*. Springer, 2006.

[23] G.V. Post. How often should a firm buy new pcs? *Commun. ACM*, 42(5):17–21, 1999.

[24] M. Baker, G. Fox, and H. Yau. A review of commercial and research cluster management software, 1995.

[25] R.L. Henderson. Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag.

[26] W. Gentzsch (Sun Microsystems). Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[27] F. Bouhafs, J.P Gelas, L. Lefèvre, M. Maimour, C. Pham, P. Vicat-Blanc Primet, and B. Tourancheau. Designing and evaluating an active grid architecture. *Future Generation Comp. Syst.*, 21(2):315–330, 2005.

[28] D.S. Myers and M.P. Cummings. Necessity is the mother of invention: a simple grid computing system using commodity tools. *J. Parallel Distrib. Comput.*, 63(5):578–589, 2003.

[29] L. Kleinrock. UCLA to be first station in nationwide computer network. available from http://www.cs.ucla.edu/ lk/REPORT/press.htm, July 1969. UCLA Press release.

[30] L. Smarr and Ch.E. Catlett. Metacomputing. *Communications of the ACM Digital library*, 35(6):44–52, June 1992.

[31] G. von Laszewski and K. Amin. *Grid Middleware*, chapter Chapter 5 in Middleware for Communications, pages 109–130. Wiley, 2004.

[32] I. Foster and C. Kesselmann. *The Grid, Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, 1999.

[33] P.V. Coveney, J. Chin, M.J. Harvey, and Sh. Jha. Scientific grid computing: The first generation. *Computing in Science and Engg.*, 7(5):24–32, 2005.

[34] R.P. Bruin, M.T. Dove, M. Calleja, and M.G. Tucker. Building and managing the eminerals clusters: A case study in grid-enabled cluster operation. *Computing in Science and Engg.*, 7(6):30–37, 2005.

[35] M. Alfredsson, E. Artacho, M. Blanchard, J.P. Brodholt, C.R.A. Catlow, D.J. Cooke, M.T. Dove, Z. Du, N.H. de Leeuw, A. Marmier, S.C. Parker, G.D. Prie, J.M.A. Pruneda, W. Smith, I Todorov, K. Trachenko, and K. Wright. eMinerals: Science outcomes enabled by new grid tools. In *Proceedings of the UK e-Science All Hands Meeting*, pages 788–795, September 2005.

[36] P. Eerola, T. Ekelof, M. Ellert, J.R. Hansen, A. Konstantinov, B. Konya, J.L. Nielsen, F. Ould-Saada, O. Smirnova, and A. Waananen. The nordugrid architecture and tools, 2003.

[37] D.A. Reed. Grids, the teragrid, and beyond. *Computer*, 36(1):62–68, 2003.

[38] LHC computing grid project "quarterly status and progress report - second quarter 2005". http://lcg.web.cern.ch/LCG/PEB/Documents/LCG-ProgressReport1-02Q05_02aug05.pdf, august 2004.

[39] W.A. Ruh, W.J. Brown, and W.X. Maginnis. *Enterprise Application Integration: A Tech Brief*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[40] G. Coulson, G.S. Blair, M. Clarke, and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15:109–126, 2002.

[41] G. Alonso, editor. *Middleware 2005, ACM/IFIP/USENIX, 6th International Middleware Conference, Grenoble, France, November 28 - December 2, 2005, Proceedings*, volume 3790 of *Lecture Notes in Computer Science*. Springer, 2005.

[42] J.C. Cunha, O.F. Rana, and P.D. Medeiros. Future trends in distributed applications and problem-solving environments. *Future Gener. Comput. Syst.*, 21(6):843–855, 2005.

[43] P. Grace, G. Coulson, G.S. Blair, and B. Porter. Deep middleware for the divergent grid. In *Middleware*, pages 334–353, 2005.

[44] R.P. Gabriel. LISP: Good news, bad news, how to win big. *AI Expert*, 6(6):30–39, jun 1991.

[45] J. Chin and P. V. Coveney. Towards tractable toolkits for the grid: a plea for lightweight, usable middleware. Technical report, National e-Science Centre, February 2004.

[46] S. Smallen, W. Cirne, J. Frey, F. Berman, R. Wolski, M. H. Su, C. Kesselman, S. Young, and M. Ellisman. Combining workstations and supercomputers to support grid applications: The parallel tomography experience. In *Proceedings of the 9th Heterogenous Computing Workshop*, May 2000.

[47] T. Hey and A. Trefethen. e-science and its implications. *Philosophical Transactions of the Royal Society of London Series A-Mathematical Physical and Engineering Sciences*, 361(1809):1809–1825, August 2003.

[48] I. Foster and C. Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(5–6):607–621, 1999.

[49] The Global Grid Forum. The globus project. http://www.globus.org. Argonne National Laboratory, USC Information Sciences Institute.

[50] M. Draoli, C. Gaibisso, and D. Giannelli. Deploying the globus security infrastructure in a production environment: Testing and evaluation. In *EuroWeb*, 2002.

[51] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

[52] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.

[53] Globus components in action. http://hpc.doc.ic.ac.uk/PPS/globus4/sld037.htm. Tutorial for the Globus Toolkit.

[54] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *NPC*, pages 2–13, 2005.

[55] S. Yang, M Hayes, K.W. Jenkins, and S. Cant. The cambridge CFD grid for large-scale distributed CFD applications. *Future Generation Comp. Syst.*, 21(1):45–51, 2005.

[56] M. Livny and M. Solomon. Condor - high throughput computing. Online available.

[57] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[58] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[59] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[60] J. Pytlinski, L. Skorwider, V. Huber, M. Wronski, and P. Bala. UNICORE - An Uniform Platform for Chemistry on the Grid. *Journal of Computational Methods in Science and Engineering*, 2(3s-4s):369–376, 2002.

[61] M. Romberg. The UNICORE grid infrastructure. *Special Issue on Grid Computing of Scientifc Programming Journal*, 10:149 – 157, 2002.

[62] D. Breuer, D. Erwin, D. Mallmann, R. Menday, M. Romberg, V. Sander, B. Schuller, and Ph. Wieder. Scientific computing with UNICORE. In D. Wolf, G. Münster, and M. Kremer,

editors, *NIC Symposium*, volume 20 of *NIC series*, pages 429 – 440, Forschungszentrum Jülich, February 2004.

[63] Unicore. http://www.unicore.eu. Project Homepage.

[64] D. Breuer, P. Wieder, S. van den Berghe, G. von Laszewski, J. MacLaren, D. Nicole, and H.C. Hoppe. A UNICORE globus interoperability layer. *Computing and Informatics*, 21:399 – 411, 2002.

[65] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. Netsolve: Grid enabling scientific computing environments. *Grid Computing and New Frontiers of High Performance Processing*, Advances in Parallel Computing, 14.

[66] Netsolve. http://icl.cs.utk.edu/netsolve/index.html. Project Homepage.

[67] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. Technical Report CS-96-328, Knoxville, TN 37996, USA, 1996.

[68] J.S. Plank, H. Casanova, J. Dongarra, and T. Moore. Netsolve: An environment for deploying fault-tolerant computing. In *FastAbstracts Session, FTCS-28: 28th International Symposium on Fault-tolerant Computing*, Munich, June 1998.

[69] H. Hussain-Khan and O. Michielin. XGrid, a "just do it"grid solution for non it's. *EMBnet.news*, 11(3):13–19, September 2005.

[70] R. Sirvent, A. Merzky, R.M. Badia, and T. Kielmann. GRID superscalar and SAGA: forming a high-level and platform-independent grid programming environment. In *CoreGRID Integration WorkShop 2005*, 2005.

[71] H.S. Sarjoughian, B.P. Zeigler, and S. Park. Collaborative distributed network system: a lightweight middleware supporting collaborative DEVS modeling. *Future Gener. Comput. Syst.*, 17(2):89–105, 2000.

[72] C. Gaspar, M. Dönszelmann, and Ph. Charpentier. DIM, a portable, light weight package form information publishing, data transfer and inter-process communication. *Computer Physics Communications*, 140:102–109, 2001.

[73] F. Curbera, M.J. Duftler, R. Khalaf, W.A. Nagy, N. Mukhi, and S. Weerawarana. Colombo: lightweight middleware for service-oriented computing. *IBM Syst. J.*, 44(4):799–820, 2005.

[74] P.V. Coveney, J. Suter, R. Saksena, L. Pedesseau, J. Blower, and E. Auden. Usable middleware for grid based computational science. presented at the e-Science Usability Meeting at NeSC, January 2006.

[75] M Hayes, L. Morris, R. Crouchley, D. Grose, T. van Ark, R. Allan, and J.M. Kewley. GROWL: A lightweight grid services toolkit and applications. In Simon Cox and David W Walker, editors, *Proceedings of the UK e-Science All Hands Meeting*, Nottingham, UK, 2005.

[76] J.D. Blower, A.B. Harrison, and K. Haines. Styx grid services: Lightweight, easy-to-use middleware for scientific workflows. In *International Conference on Computational Science (3)*, pages 996–1003, 2006.

[77] P.V. Coveney, R.S. Saksena, S.J. Zasada, M. McKeown, and S. Pickles. The application hosting environment: Lightweight middleware for grid-based computational science, 2006.

[78] S. Benkner, I. Brandic, G. Engelbrecht, and R. Schmidt. VGE - a service-oriented grid environment for on-demand supercomputing. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 11–18, Washington, DC, USA, 2004. IEEE Computer Society.

[79] Sun Microsystems, Inc. RFC 1057: RPC: Remote procedure call protocol specification: Version 2, jun 1988.

[80] R. Srinivasan. RFC 1831: RPC: Remote procedure call protocol specification version 2, aug 1995. Status: PROPOSED STANDARD.

[81] S.S. Vadhiyar and J. Dongarra. GrADSolve: a grid-based RPC system for parallel computing with application-level scheduling. *J. Parallel Distrib. Comput.*, 64(6):774–783, 2004.

[82] C. C. Chang, G. Czajkowski, and T. Von Eicken. MRPC: A high performance RPC system for MPMD parallel computing. *Concurrency - Practice and Experience*, 29(1):43–66, 1999.

[83] H. Nakada, S. Masuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A gridRPC model and API for end-user applications. Technical report, GridRPC Working Group, 2005.

[84] M. Hirano, M. Sato, and Y. Tanaka. OpenGR: a directive-based grid programming environment. *Parallel Comput.*, 31(10-12):1140–1154, 2005.

[85] P.V. Coveney, J. Vicary, J. Chin, and M. Harvey. WEDS:a web services-based environment for distributed simulation. *Phil. Trans. R. Soc. A.*, (363):1807–1816, 2005.

[86] S. Benkner et.al. VGE - the vienna grid environment. http://www.par.univie.ac.at/project/vge/.

[87] G. Tsouloupas and M. Dikaiakos. Design and implementation of gridbench. In *Advances in Grid Computing - European Grid Conference 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 211–225, Amsterdam, The Netherlands, June 2005. Springer. Revised Selected Papers.

[88] Cpan. http://www.cpan.org.

[89] R. Orfali, D. Harkey, and J. Edwards. *The essential client/server survival guide (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[90] W. Zhou. Supporting fault-tolerant and open distributed processing using RPC. *Computer Communications*, 19(6-7):528–538, 1996.

[91] Ch. Ede, J. Schweifer, and A. Scrinzi. An HTML-based grid-interface for computational photonics application. Technical report, Photonics Institute, Vienna University of Technology, Gusshausstrasse 27/387, A-1040 Vienna, December 2006.

[92] K.W. Umbach. What is "push technology"? Technical Report 6, California Research Bureau, California State Library, October 1997. CRB Note.

[93] Anonymous and Sams Development Staff. *Maximum Security: A Hacker's Guide to Protecting Your Internet Site and Network, 2nd Edition*. Sams, Indianapolis, IN, USA, 1998.

[94] J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[95] Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001.

[96] P. Chown. AES ciphersuite for TLS. Internet draft by the Network Working Group, October 2000.

[97] J. Schweifer and K. Schwarz. W2GRID - users guide. online available at (http://www.w2grid.at/downloads/usersguide.pdf), 2006.

[98] U. Schwiegelshohn, P. Wieder, and R. Yahyapour. Resource management for future generation grids. volume TR-0005 of *CoreGRID Technical Report*, May 2005.

[99] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.

[100] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[101] L. Prechelt. Are scripting languages any good? A validation of perl, python, rexx, and tcl against C, C++, and Java. *Advances in Computers*, 57:207–271, 2003.

[102] S. Tregar. *Writing Perl Modules for CPAN*. APress, New York, NY, 1st edition, 2002.

[103] J. Schweifer and K. Schwarz. W2GRID - developers guide. will soon be available online at (http://www.w2grid.at/downloads/developersguide.pdf), 2006.

[104] J.T. Chiou, Ch. Changli Chin, and S.R. Tsai. A fault tolerant RPC mechanism based on ip multicasting. *J. Syst. Archit.*, 43(10):701–717, 1997.

[105] G. von Laszewski. The grid-idea and its evolution. *it - Information Technology*, 47(6):319–329, 2005.

[106] I. Foster. What is a grid? a three point checklist. *Grid Today*, 1(6), July 2002.

[107] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 225, Washington, DC, USA, 2002. IEEE Computer Society.

[108] A. Barmouta and R. Buyya. Gridbank: A grid accounting services architecture (GASA) for distributed systems sharing and integration. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 245.1, Washington, DC, USA, 2003. IEEE Computer Society.

CURRICULUM VITAE

Johannes M. Schweifer

ADDRESS

Aspangstrasse 51/27
1030 Wien, Austria
Phone: +43-1-58801-15672
Fax: +43-1-58801-15698
Email: jsch@mail.zserv.tuwien.ac.at

PERSONAL DATA

Sex: male
Date of Birth: Eisenstadt, Ausria
Parents: Johann and Waltraud Schweifer
Citizenship: Austria

EDUCATION

1982-1986	Elementary School in St. Margarethen
1986-1995	Secondary School in Mattersburg
1995-2002	Study of Chemical Engineering at the Vienna University of Technology finished with honor on the 14.1.2002

PROFESSIONAL APPOINTMENTS

1999-2001	Researcher in the group of Prof. Wolgang Linert at the Institute of Inorganic Chemistry within the TMR/TOSS project.
2000-2001	employed as tutor at the institute of Inorganic Chemistry
2002	Project assistant in the group of Prof. Walter Lengauer at the Institute of Chemical Technologies and Analytics
2003-	Researcher in the group of Prof. Karlheinz Schwarz at the Institute for Materials Chemistry within the AURORA project.

PUBLICATIONS

1. **The W2GRID middleware plugin for WIEN2k**
   J. Schweifer and P. Blaha and K. Schwarz
   to appear in: "2nd Austrian Grid Symposium", J. Volkert, T. Fahringer, D. Kranzlmüller, W. Schreiner (Hrg.); Österreichische Computer Gesellschaft

2. **"From crystal structure to properties of solids with the grid-enabled WIEN2k"**
   K. Schwarz, P. Blaha, J. Schweifer
   in: "1st Austrian Grid Symposium", J. Volkert, T. Fahringer, D. Kranzlmüller, W. Schreiner (Hrg.); Österreichische Computer Gesellschaft, 2006, 3-85403-210-2, S. 25 - 35

3. **"Competing structural instabilities in the ferroelectric Aurivillius compound $SrBi_2Ta_2O_9$"**
   J.M. Perez-Mato, M. Aroyo, A. Garcia, P. Blaha, K. Schwarz, J. Schweifer, K. Parlinski
   Physical Review B, 70 (2004), S. 214111

4. **"Synthesis and characterisation of tetrazole compounds: 3 series of new ligands representing versatile building blocks for iron(II) spin-crossover compounds"**
   M. Grunert, P. Weinberger, J. Schweifer, C. Hampel, A. Stassen, K. Mereiter, W. Linert
   Journal of Molecular Structure, 733 (2005), S. 41 - 52

5. **"Structure and Physical Properties of [μ-Tris(1,4-bis(tetrazol-1-yl)butane - N4,N4')iron(II)] Bis(hexafluorophosphate), a New Fe(II) Spin-Crossover Compound wizh a Three-Dimensional Threefold Interlocked Crystal Lattice"**
   M. Grunert, J. Schweifer, P. Weinberger, W. Linert, K. Mereiter, G. Hilscher, M. Müller, G. Wiesinger, P. van Konigsbruggen
   Inorganic Chemistry, 43 (2004), S. 155 - 165

6. **"Catena [μ-Tris(1,2-bis(tetrazol-1-yl)ethane-N4,N4')iron(II)] bis(tetrafluorborates): Synthesis, Structure, Spectroscopic and Magnetic Characterization of a chain-type Coordination Polymer Spin-Crossover Compound"**
   J. Schweifer, P. Weinberger, K. Mereiter, M. Boca, C. Reichl, G. Wiesinger, G. Hilscher, P. van Konigsbruggen, H. Kooijman, M. Grunert, W. Linert
   Inorganica Chimica Acta, 339 (2002), S. 297 - 306

7. **"Metallomesogenic β-diketone and salicylaldimine oxovanadium(IV)complexes"**
   O. Costisor, J. Schweifer, M. Grunert, W. Linert
   in: "Recent Research Developments in Materials Science", Transworld Research Network, Trivandrum, Kerala, Indien, 2001, ISBN 81-7736-045-0, S. 1 - 18

PRESENTATIONS

1. **The W2GRID middleware plugin for WIEN2k**
   2^{n}d Austrian Grid Symposium, Innsbruck; 22.09.2006

2. **W2GRID - a perl based middleware**
   Dept. Earth Sciences, University Cambridge, Cambridge, UK; 13.09.2006

3. **W2GRID in two applications"**
   13^{t}h Aurora Plenary Meeting, Strobl/Wolfgangsee; 09.06.2006 - 11.06.2006

4. **W2GRID, a generic Grid infrastructure**
   Scientifi c Grid-Computing, TU Wien; 22.04.2005

5. **Functionality of W2GRID**
   9^{t}h AURORA Plenary Meeting, Strobl/Wolfgangsee; 04.06.2004 - 06.06.2004

POSTERS

1. **W2GRID, a grid computing infrastructure for WIEN2k**
   J. Schweifer, K. Schwarz, P. Blaha
   $19^t h$ Workshop on Novel Materials and Superconductivity, Planneralm; 22.02.2004 - 28.02.2004

Wien, am 14. Dezember 2006