NOMADIC MIGRATION – A SERVICE ENVIRONMENT FOR AUTONOMIC COMPUTING ON THE GRID

von

Gerd Lanfermann

vorgelegt der

Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

im November 2002

zur Erlangung des Akademischen Grades
Doktor der Naturwissenschaften
– Dr. rer. nat. –

angefertigt am

Max-Planck-Institut für Gravitationsphysik, Potsdam
Albert-Einstein-Institut

und

Fachbereich Informatik, Universität Potsdam

Gutachter:

Professor Dr. Bettina Schnor
Professor Dr. Edward Seidel
Professor Dr. Ian Foster

Abstract

In recent years, there has been a dramatic increase in available compute capacities. However, these “Grid resources” are rarely accessible in a continuous stream, but rather appear scattered across various machine types, platforms and operating systems, which are coupled by networks of fluctuating bandwidth.

It becomes increasingly difficult for scientists to exploit available resources for their applications. We believe that intelligent, self-governing applications should be able to select resources in a dynamic and heterogeneous environment: Migrating applications determine a resource when old capacities are used up. Spawning simulations launch algorithms on external machines to speed up the main execution. Applications are restarted as soon as a failure is detected. All these actions can be taken without human interaction.

A distributed compute environment possesses an intrinsic unreliability. Any application that interacts with such an environment must be able to cope with its failing components: deteriorating networks, crashing machines, failing software. We construct a reliable service infrastructure by endowing a service environment with a peer-to-peer topology. This “*Grid Peer Services*” infrastructure accommodates high-level services like migration and spawning, as well as fundamental services for application launching, file transfer and resource selection. It utilizes existing Grid technology wherever possible to accomplish its tasks. An Application Information Server act as a generic information registry to all participants in a service environment.

The service environment that we developed, allows applications e.g. to send a relocation requests to a migration server. The server selects a new computer based on the transmitted resource requirements. It transfers the application’s checkpoint and binary to the new host and resumes the simulation. Although the Grid’s underlying resource substrate is not continuous, we achieve persistent computations on Grids by relocating the application. We show with our real-world examples that e.g. a traditional genome analysis program can be easily modified to perform self-determined migrations in this service environment.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Professor Edward Seidel, for his guidance and encouragement through the past years (and two degrees). I am truly fortunate to have been one of his students; he gave me the intellectual freedom to define the path of my research in Physics and Computer Science. He provided me with the support to complete this work.

I would like to thank Professor Bettina Schnor for the valuable insights and inputs for a computer science “quereinsteiger” like me. I have greatly benefited from the discussions with her and her research group.

My gratitude to Professor Ian Foster, who served on my thesis committee. He supported a research stay at the University of Chicago, which became a source of ideas for this work.

I also wish to thank Dr. Gabrielle Allen for her advice and support during my experiments. Special thanks go to my colleague Thomas Dramlitsch, our local Globus guru.

Important were the valuable discussions with the members of the Cactus Group and the GridLab team at the Albert Einstein Institute, in particular Tom Goodale, Jason Novotny, David Rideout, Thomas Radke, Kashif Rasul, Michael Russel and Oliver Wehrens. I have benefited from many conversations with Dave Angulo at the University of Chicago and Andre Merzky at the Konrad Zuse Institute.

This work was supported by the EU-GridLab project, the GrADS project and the Max Planck Institute for Gravitational Physics, Albert Einstein Institute. Computing Resources and technical support have been provided by AEI, NCSA, University of Potsdam and the European Grid. I’m grateful for the use of their resources.

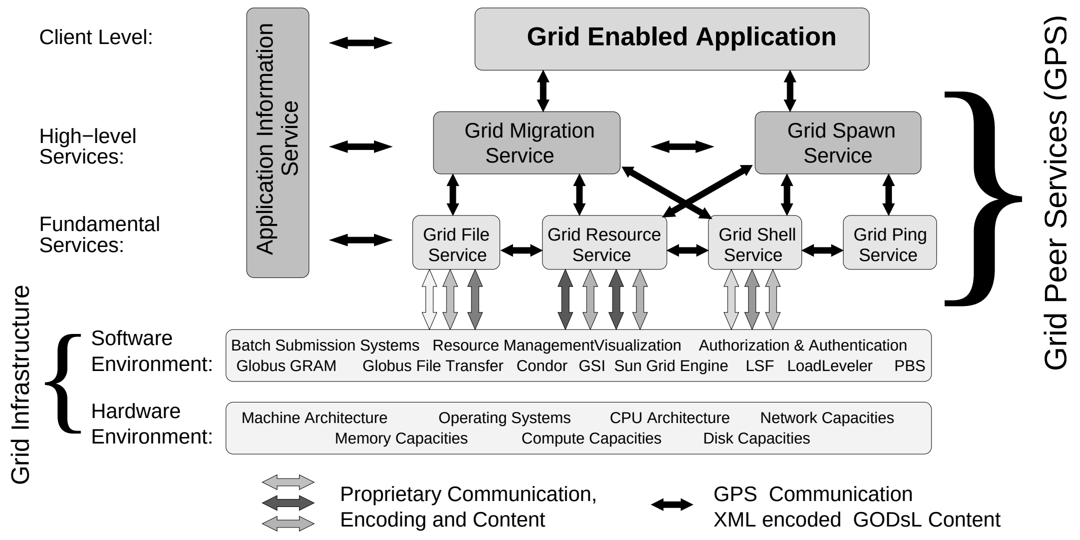
Chapter 1

Introduction and Overview

Nomadic tribes move their entire possessions to a new dwelling, when a new habitat promises more efficient hunting and gathering. We believe that applications on a Grid can operate with similar autonomy, when they seek out new compute capacities once the old resources are used up or when they restart on a new host if the previous hardware fails.

Such a migration technology must be intelligent, failure proof and it must respect the heterogeneity of multi-organizational Grids. It involves tasks like resource monitoring, application and service discovery, as well as file transfers of arbitrary size. The objective of this thesis is the design and implementation of a generic, fault-tolerant service infrastructure that allows self-determined operations like migration on heterogeneous Grids.

We start in Chapter 2 with a brief introduction to Grid computing, followed by a presentation of three advanced case studies to highlight the potential of autonomic computing on Grids. Two of these examples, which are motivated by previous prototypes [4, 60], migration and spawning, are implemented in this work. We continue the chapter with an analysis of the characteristics of a Grid environment. Based on our experiences [9], we focus on the heterogeneity of the different Grid components and the intrinsic unreliability in Grids. We conclude Chapter 2 by outlining the aims of this thesis. Existing Grid technology and related research is reviewed in Chapter 3. Wherever possible, we use the existing infrastructure in our service infrastructure.



Client Level:
High−level Services:
Fundamental Services:
Application Information Service
Grid Enabled Application
Grid Migration Service
Grid Spawn Service
Grid File Service
Grid Resource Service
Grid Shell Service
Grid Ping Service
Grid Peer Services (GPS)
Grid Infrastructure
Software Environment:
Hardware Environment:
Batch Submission Systems Resource Management Visualization Authorization & Authentication
Globus GRAM Globus File Transfer Condor GSI Sun Grid Engine LSF LoadLeveler PBS
Machine Architecture Operating Systems CPU Architecture Network Capacities
Memory Capacities Compute Capacities Disk Capacities
Proprietary Communication, Encoding and Content
GPS Communication XML encoded GODsL Content

Figure 1.1: The layer diagram gives an overview of the work in this thesis and outlines the embedding of the Grid Peer Services (GPS) between user application and Grid infrastructure. High-level services are built from fundamental services, which use existing Grid technology. Redundant deployment of services prevents single points of failure. GPS communication content is expressed through Grid Objects (GODsL).

Based on our analysis of the Grid characteristics, we propose in Chapter 4 a pseudo-reliable service infrastructure by combining the web service model with a peer-to-peer topology [61]. We call this fusion of these two service paradigms in a Grid environment the *Grid Peer Service* model. In Chapter 5, we motivate a generic information model that allows a precise characterization of objects on a Grid, e.g. files on machines, services in applications, compute requirements of simulations. We use this *Grid Object Description Language* [62] to integrate existing Grid technology as well as legacy applications into our service environment.

The service components of our infrastructure are built around a client-server framework, explained in Chapter 6, where we implemented a request handler to experiment with different error propagation strategies and thin-client design. In Chapter 7 we introduce our implementation of fundamental services, as illustrated in Figure 1.1. They allow us to construct the compound migration and spawn services. These services are treated in Chapter 8, with special emphasis on failure strategies and reliable client startup.

In Chapter 9 we demonstrate the capabilities of the migration and spawn service by migrating, auto-recovering and spawning real-world simulations in the field of numerical relativity [6, 1] and genome analysis [63]. Both applications have large requirements for memory, processors and compute time. We conclude this thesis with a summary and an outlook on future projects in Chapter 10.

Chapter 2

Using Computational Grids

Computational simulations have always complemented theoretical and experimental research. In the future, more disciplines than ever before rely on computational approaches to verify and validate scientific results: Weather forecasting, genome analysis and financial risk management are just a few of the many fields which increasingly depend on high performance computing.

However, scope and accuracy of these computations are limited by the computational hardware. In quite a few situations, these limits can be overcome by using multiple networked computers or by deploying intelligent programs that seek out and acquire new resources if their current compute capacity proves insufficient.

We begin this chapter with an overview of Grid computing in Section 2.1. We continue with the introduction of three advanced Grid case studies in Section 2.2, which portray the potential power of global Grids. In Section 2.3 we confront the reader with the severe problems that are deeply rooted in global, heterogeneous Grids: We examine its characteristics, highlight the software and hardware aspects and analyze the problems imposed by unreliable networks. We conclude the chapter in Section 2.4 with the objective of this thesis: based on our analysis, we propose a *service topology* and an *information model* to create a service infrastructure which operates fault-tolerantly and interfaces with the ubiquitous, legacy Grid middleware in heterogeneous Grids.

2.1 Introduction to Grid Computing

Grid computing originated within the scientific and technical computing segment and describes the ability to access and use distributed computer resources independently of scale, hardware and software. The term "Grid" was inspired by the analogy to power grids [39], which give people access to electricity, while the location of the electric power source is far away and usually completely unimportant to the consumer. The power sources can be of different type, burning coal or gas or using nuclear fuel, and of different capacity. All of these characteristics are completely hidden to the consumer, who only experiences the electric power, which he can tap out of sockets, using commodity equipment like plugs and cables.

While the advantages of computing grids are obvious, no exact definition of "the Grid" exists. An early definition reveals the similarities to the Power Grid analogy [36]: *"A computational grid is a hardware or software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities."* The current trend seems to have dropped the "high-end" and promotes Grids for every hardware level and type.

Grid computing has been a buzzword over the last years and computer science institutes have taken great effort in providing software packages to create a Grid infrastructure. The Grid "hype" has been amplified by industrial companies like IBM and SUN moving in and picking up on this trend. Still the number of *applications*, which utilize Grids across multiple organizations remains small. With the wealth of more or less useful Grid infrastructure in place, it is crucial to establish the *application* as the true driving force of Grid development. It is important to keep focused on possible application scenarios as a common goal for both application programmers and computer scientists. Therefore we describe a number of these scenarios, which are based on our personal experience with large scale simulation in the field of numerical relativity.

2.2 Advanced Grid Usage Scenarios

This section describes three advanced Grid usage scenarios. Turning them into reality was the main motivation for conducting the research presented in this thesis. Two of these case studies have been carried out and are and described in Chapter 9. Early Grid computing experiments like using two or more supercomputers as a single virtual machine, called a "*meta-computer*", date back to even before 1992, when Smarr and Catlett outlined such an approach [84]. This thesis focuses on a different type of meta-computing: we are concerned with the autonomic execution of (parallel) programs across various machines [9, 60], which we term *"Nomadic Migration"*.

2.2.1 Nomadic Migration

Nomadic Migration defines the self-controlled and automatic relocation of large applications to provide faster or more efficient code execution. *Nomadic* illustrates the low frequency of this event and the self-contained operation of the application: It is an application-initiated and self-determined process in a service environment. It handles large, parallel simulations but is not intended to be used as a fast reacting load-balancing system.

Migration and its strategies were well studied in the context of cycle-stealing clusters in the mid 90s (see for example [73, 81, 15]). The motivation for migration in these environments and on Grids has more or less remained the same:

- Administrative reasons: The compute requirements of an application goes way beyond the queue time limits offered at a compute center.
- Performance reasons: A "better" compute capacity becomes available during the execution and a higher throughput is achieved if the job is relocated.

In this thesis we go beyond the scope of a migrating within a single cluster or an intra-Grid machine pool, but relocate across global Grids. The next paragraph illustrates the advantages of automatic migration over manual transfer of applications.

Traditional Job Management: Many large scale simulations have compute time requirements which go way beyond the queue time limits offered at supercomputing centers. There are even cases where the application's runtime cannot be predicted at the start of the simulation. The requirements for other resources like memory or disk space may also change during the course of the simulation, sometimes exceeding the supply.

If a user is confronted with resource demands by his applications that exceed the current setting, but he wants to continue the simulation anyway, he has to relocate to a machine or batch queue that provides the required resources. The user engages in a tedious process of instructing the application to checkpoint, securing the checkpoint files and archiving them if necessary. If the simulation can only be continued on another host, the checkpoint file needs to be transferred. After deriving the new resource requirements the simulation is resubmitted to the queuing system. At all steps, the user's involvement makes the process prone to failure:

1. Checkpoint files can be erased by disk quota expiration if the user does not move the data in time.

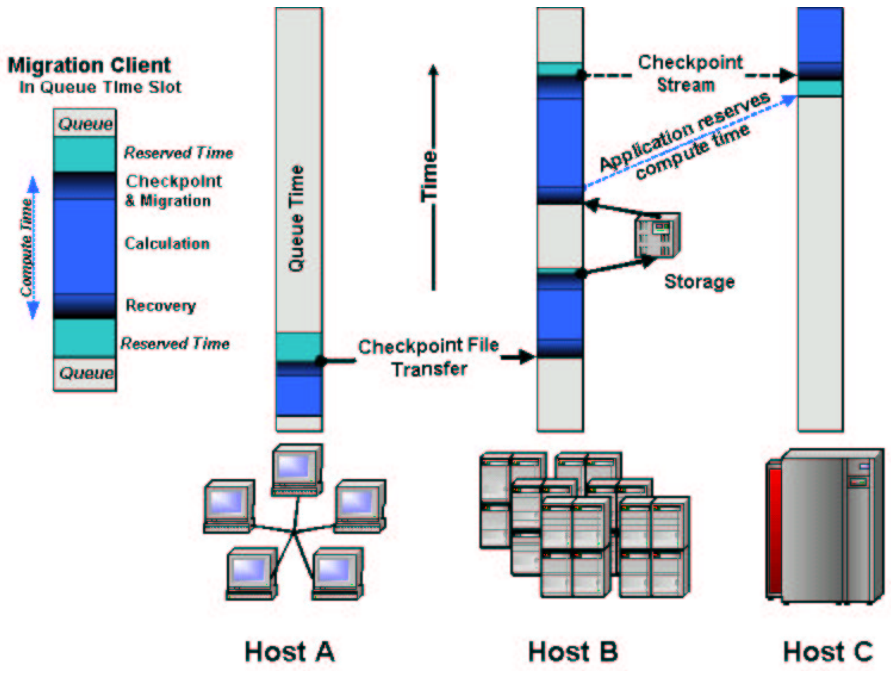


Figure 2.1: “Nomadic Migration” describes the process of interrupting a large-scale application and continuing it on a different machine through the transfer of checkpoints.

2. Resource requirements have to be correctly analyzed, or the job will be rejected from the resource management system. This may happen either at submission time or – even worse – during runtime.

3. Conservative resource estimates lead to longer waiting times in the queue.

4. The researcher is required to remember usernames and passwords as well as the interfaces to a wide range of different machines, architectures, batch systems and shell programs.

From experience, the overhead of this procedure encourages the researcher to resubmit a job on the same machine, discarding potentially faster machines.

Automated Migration: The process of submitting an application on an arbitrary host, which fulfills minimum resource requirements, is a prime candidate for automation. In this section we describe the migration service in general and draw the attention to some of the problems that have to be solved.

In Figure 2.1 we illustrate a migration process for an application over time. The migration progresses from left to right across three different host types *A*, *B* and *C*, indicating a network of workstations, a cluster and a traditional supercomputer, respectively. The left inset shows the different phases of an application in a queue: a migration client receives reserved compute time, called a “slot”. Within this time slot, all code execution has to take place: the recovery of the simulation state from a checkpoint (after the first migration), the calculation phase and the process of writing the simulation state to a checkpoint file. Applications which exceed their time slots are terminated.

In the illustration the simulation is started on host *A*. A migration server (not shown), which is responsible for the relocation of applications, receives information on the simulation’s resource consumption and its location. It monitors the availability of new machines which meet the simulation’s requirement. As “better” resources become available, the simulation on *A* is informed and it checkpoints. The checkpoint files are transferred to host *B*. The simulation is restarted or submitted to the queuing system. As the application runs out of compute time on *B*, the last checkpoint is archived in a storage facility and the simulation is resubmitted to the queue on the same machine. The simulation will execute a second time on *B*.

Some advanced applications are able to receive the checkpoint as a socket stream instead of reading from file. In combination with advanced reservation scheduling, this transfer mode allows for fast checkpoint transfer, shown in the migration to machine *C*: The application on *B* is aware of the expiration of its queue time and requests in advance a slot on machine *C*, which overlaps with the compute slot on *B*. By the time the application is about to finish on *B*, the migration server starts an uninitialized simulation on *C*, which receives the simulation state through the streamed checkpoint and continues the calculation.
This case study identifies some of the problems that have to be solved:

- *File Transfer*: A migration server stages executables and other files to a new host. Therefore, a migration server must be able to determine and use the different file transfer methods for each of the source and target machines.
- *File Description*: A service must provide a compact description of multiple files, including information on where the files are located and how they can be accessed.
- *Job Submission:* The migration server is responsible for submitting the new job to the individual queuing systems.
- *Resource Description*: It is essential to characterize resource requirements of an application as well as resource capacities of machines. The server must be able to evaluate and compare them.
- *Hardware Description*: A migration service must describe the location of machines, including the ways to access it.
- *Fault Tolerance*: Any participating service must operate in a fault tolerant way. If an application requires a service, it must be able to determine a service instance somewhere.

Nomadic migration offers a way to radically increase the throughput of long-term simulations by automating the resource selection and application transfer process.

2.2.2 Application Spawning

Application Spawning is a strategy to speed-up a simulation by taking advantage of work flow parallelism and out-sourcing parts of an application to external hosts. Many types of simulations usually involve a chain of different algorithms, whose outcome may feed back into the main simulation. The result of a computation, e.g. an analysis routine, may not be needed for some time. We can perform this calculation on a different computer, outside the main program flow. However, we have to make sure that the data is sent back in time for the main calculation to continue. We call this technique *application spawning*. In the following scenario, we focus on the spawning of algorithms, which do not provide data feedback to the core simulation. In Figure 2.2 we illustrate the splitting of a simulation work flow into a sequence of core algorithms *A,C,E,...* and a set of sub-algorithms *B,D,F,H,...* that are spawned.

Each sub-algorithm is executed in a standalone executable on a machine which is “best”-qualified for that type of task. The application determines autonomically at runtime, which sub-routines in its work flow do not feed back immediately into the main simulation and can be spawned. The simulation

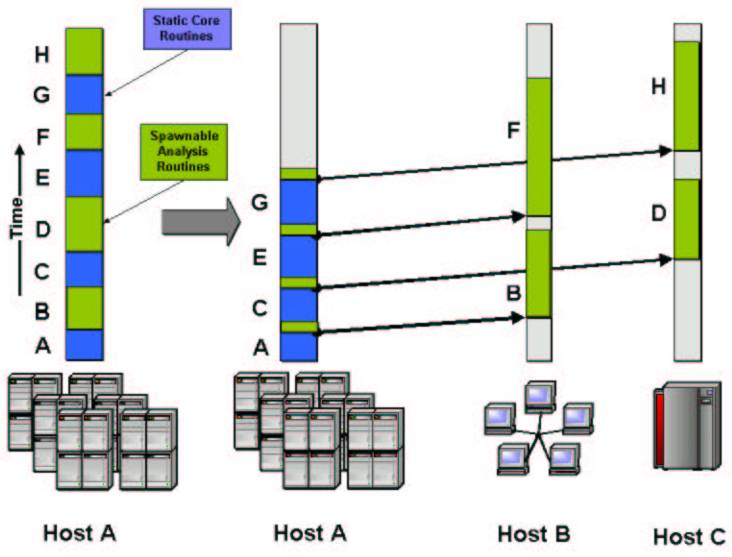


Figure 2.2: *Application spawning* describes a technique to identify regions in the program's work flow that are parallel to the main execution and can be run on external resources. Spawning customizes an application to perform the resource demanding computations on expensive machines and to outsource less challenging routines to economic compute resources, like cluster pools.

writes a checkpoint which only contains the data that is necessary to start the spawned tasks. For this reason, spawned sub-jobs do not come with large checkpoints and usually run with reduced memory requirements. The hosting machine can be chosen to best match the characteristics of the spawned sub-jobs, e.g. in terms processing architecture and power, disk capacity, etc. In our illustration we execute the sub-jobs *B*, *F* on a workstation network *B* and subjobs *D*, *H* on a mainframe *C*, while the core algorithms are continued on cluster *A*.

Note that "better" resource does not necessarily mean faster, it can also be a "cheaper" machine: If timely completion of the sub-job is not an issue for the researcher, application spawning can be used to populate the inexpensive idle workstation cycles with useful computation. The high-quality compute time on a supercomputer is dedicated to the core calculations. We have gained through this approach a shorter usage of *expensive* high-performance computers by outsourcing less challenging routines to *economic* resources.

The design of a migration and spawn service environment is described in Chapter 7 and 8. Experiments with the application spawning and migration services are presented and analyzed in Chapter 9.

2.2.3 Simulation Prototyping

Simulation Prototyping allows a simulation to determine its *future* numerical behavior by launching simulation probes. These probes "overtake" the main execution and their behavior is used to control the principal simulation.

Simulation prototyping is useful for any simulation that depends on the setting of certain parameters, e.g. Adaptive Mesh Refinement (AMR) algorithms. AMR methods smooth out critical regions in numerical simulations by locally increasing the resolution of the mesh. The process of adding more points to the mesh and choosing a smaller time step is called *mesh refining*. Certain local numerical conditions, (e.g. an increase in error residuals during the evolution process) may indicate a critical area, which requires higher spatial refinement. Often these conditions do not give enough information

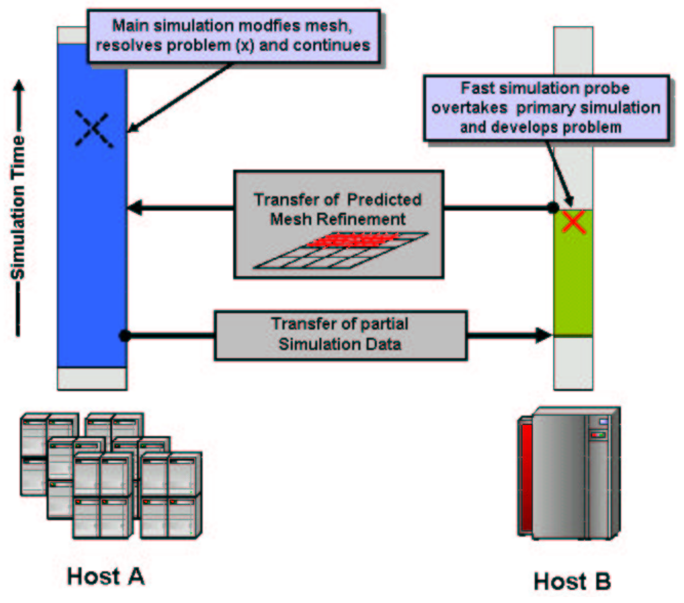

Main simulation modfies mesh, resolves problem (x) and continues
Fast simulation probe overtakes primary simulation and develops problem
Simulation Time
Transfer of Predicted Mesh Refinement
Transfer of partial Simulation Data
Host A
Host B

Figure 2.3: Simulation Prototyping denotes fast-running probes that explore the behavior of a parent simulation. In this illustration, an AMR simulation launches a probe to detect problematic regions in advance.

on the exact location of a region. Since adding mesh points implies a serious increase in memory consumption, the refinement of the whole mesh is generally not an option.

A solution to this problem is illustrated in Figure 2.3, where a simulation probe is started on host *B*. The probe only contains parts of the full problem: it uses the same numerical algorithms but operates with lesser precision, resolution and spatial extent. Because the probe on host *B* executes faster, it "overtakes" the parent simulation on host *A* in time and develops the undesired pathological behavior earlier. The probe's simulation is stopped, the results are transferred to the principal simulation and are used to identify the critical regions in advance. The right number of mesh points is added to the main simulation before the pathology develops.

A variation of the approach is used to conduct parameter studies: a numerical model often depends on many different physical parameters. To gain an understanding of their impact, many simulations ("clones") with slightly different parameter settings are launched to survey the parameter space. Information of such a survey feeds back into a parent simulation.

While this particular scenario is not implemented in this thesis, we gave the the reader an impression of the possibilities that are offered by intelligent applications. It will become evident at the end of this thesis how such a self-controlled application can be realized in our Grid Peer Service environment. With the realization of migration and spawning techniques in this thesis, the possibilities of autonomic Grid computing may appear less imaginary.

2.3 Characteristics of a Grid Environment

In this section we analyze the characteristics of Grids. We concentrate on networks in Section 2.3.1, the hardware environment in Section 2.3.2 and software environment 2.3.3 as the core components of Grids. We restrict our analysis to multi-organizational, global computing Grids and use our observations later in the design of a service infrastructure. In Section 2.3.4 we derive requirements for applications to run and migrate in a heterogeneous Grid environment. We look in particular at the hardware independence of the executable and file data formats.

The characteristics of Grids are largely a result of a *heterogeneous resource substrate* [26] combined with *autonomous site administration*. Resources are typically owned and operated by different organizations with various site policies and application backgrounds. The term *substrate* portrays the underlying hardware, network, software, compiler and resource management layer, which is different from site to site and varies with time. Even if the same components are employed at different sites, differences in the local configuration settings are usually quite large. Obviously the substrate is not static but has dynamic properties: the components of a Grid change some of its properties over the course of months, days or hours.

2.3.1 Networks and Reliability

Computational Grids can be realized on a large range of settings and we give an idea of what is typically understood by the term “Grid”:

- Combining the supercomputers of several computing centers is an example where only a few high performance computers from different organizations assemble a computing Grid. Such a collection is typically used to solve large scale numerical problems, which do not fit on a single supercomputer.

- Utilizing idle cycles of PC sized workstation: Such a network of workstations (NOW) is a viable approach to close in on the performance of a supercomputer by summing up idle workstation cycles. This collection of machines is most effectively used by *trivially parallel* algorithms.

- “Intra-Grids” are assembled from the hardware of a company’s intranet only: Such encapsulated Grids require lower security standards and ease the social aspect of co-allocating resources.

- Combining of all storage facilities into a single virtual storage space has found a popular realization in today’s Storage Area Networks (SAN). Such “storage Grids” are also subject to extensive research by the European Union’s “Data Grid” project [27], which focuses on distributed and shared large-scale databases of high-energy-physics experiments.

All these scenarios share the property that several resources are tied together by a network. Through this link, a new “virtual” resource comes to existence: a computing or storage Grid. The lifeline of such a Grid computer is the network. As the network’s quality decreases, the “Grid computer” fades out of existence. Increasing the complexity and size of the network does not make computational Grids more reliable *per se*. The network possesses an *inherently disruptive* property and even if the reliability of a single connection may be close to 100%, the chance of failure grows to significant size as more connections are added.

Distributed Applications and Networks: The dynamic bandwidth and latency of a connection on the network will vary vastly over time, as the bandwidth may be ranging several orders in magnitude. The assumption that the dynamic properties are *constant during the lifetime* of an application allows for basic performance balancing: At the start of a program, the network quality is measured and basic strategies are employed to compensate the network situation. The settings are not modified during the lifetime of the program. The initial adjustment of the TCP buffer size for Globus socket based meta-computing is an example of such a strategy.

For long-term distributed applications like services or simulations this assumption no longer holds true: the network’s dynamic property may change significantly e.g during a one week simulation run. The network quality may vary up to the degree that the network becomes unusable for certain periods.

A distributed application, which fails to acknowledge such a change in the underlying fabric, may either stop working or continue run in a rather inefficient way. The adaptability of the communication in an application is currently investigated by Dramlitsch [8, 29].

Understanding the effect of changing network quality is imperative for any service that interacts with distributed resources or applications. In this thesis we chose a distributed service environment and a redundant service topology (Chapter 4) to provide a fault-tolerant service environment to applications.

2.3.2 Grid Hardware Environments

Grids span a wide range of hardware. The variety of operating systems and hardware architectures is further inflated by the different compute capacities they offer. Certain machines are able to host larger applications, while others execute faster.

A user who is offered the heterogeneous machine park of a Grid has usually no idea how well his program will perform on the assortment of resources. On a conventional multiprocessor system, it is straightforward to derive the computational performance of an application. The efficiency on a Grid can often only be determined at runtime. Extracting this information from the application and drawing the consequences like stopping and restarting a code somewhere else puts a significant management overhead onto the user.

Computer hardware exhibits a similar unreliability as the network: machine access becomes impossible as machines are shutdown for maintenance, cluster nodes experience failures, or hard disk capacities fill up, etc. In general we have to distinguish between the machine and resource accessibility[1]. While the machine itself may be accessible, its resource is usually not immediately available: supercomputers often partition their total compute capacities into queues and execute jobs through batch submission systems. The wait time in such queues may range from hours to days.

2.3.3 Grid Software Environments

The various software environments are another factor which cause the heterogeneous appearance of Grids across multi-organizational domains. Different software packages offer solutions for secure access, batch submission and resource scheduling, file transfer, etc. The packages are tuned to perform well for a single site and are intended to be used by users – they are not designed to be utilized from external domains and in an automated fashion.

For parallel applications the different parallelization libraries are another important characteristic of a software environment: vendor implementations of MPI can be found along with generic MPI installation, like MPICH. The variety continues for the compilers, where some allow for compilations that are tightly tuned to a chip architecture, while other “just” compile an executable. In Chapter 3 we acquaint the reader with a number of different Grid software solutions when we give an overview on Grid middleware that is relevant for our migration and spawn scenarios.

Software packages are probably the most diverse but least dynamic components in a Grid. Installations are generally quite different, but they are usually not changed, once they have been adjusted to a hardware and proven their functionality.

2.3.4 Client Applications

An application needs to fulfill a number of requirements to be able to execute and migrate on a heterogeneous Grid. Beyond these minimal requirements, an application will not perform optimal

[1]By “resource” we refer to the compute capacity of one or more machines, by “machine” we denote the hardware.

unless it brings a certain “awareness” for the dynamics in a Grid. In the next section, we explain what these requirements are and how *application intelligence* improves performance in a computational Grid.

Hardware Independence: Before any applications can run across a variety of heterogeneous platforms, an appropriate binary must be supplied to the different architectures and operating systems. Several solution are feasible:

1. Hardware independence of the executable can be achieved by taking an interpretive approach, as seen with High Performance Java [56]. This strategy uses the native platform independence of Java. Although a Java implementation of MPI [14] is available, Java has not been overwhelmingly successful is numerics, since interpretive languages lack the potential of fast execution and special breeds of supercomputers lack Java runtime environments.

2. Providing the appropriate binary at each site is another strategy. For standard applications, like Mathematica or a chemical engineering software, this requirement may be easy to fulfill. If the binaries are generated by a user, this strategy does not scale well for a large number of sites: it requires an enormous amount of file management to keep binaries up-to-date on the different hosts. Feider [32] has addressed this problem with an automated make-system which compiles the source code on an arbitrary number of platforms.

3. A scalable solution can be achieved by supplying binaries for each of the participating architectures and storing them in a central repository. This solution reduces the number of binaries to the number of architectures in a Grid but possibly ignores site, compiler and chip specific optimizations.

4. Compilation-on-the-fly generates an executable from source code just in time for the execution. This approach makes it possible to use chip specific compiler optimizations, instead of using a generic executable. It requires on the other hand a very disciplined programming style, which needs to conform closely to standards like e.g. ANSI C. If the programmer has employed architecture specific optimization without proper protection, the program may perform poorly or will not compile at all.

Data Formats and Checkpoints: Platform independence is also necessary for the output data, which is generated by the application. A migrating simulation leaves a trail of output data on each host. It must be ensured to that file chunks can be joined together at a later stage. Checkpoints are data files, which are used to save the state of a simulation. They permit to restore the state of the application and continue the program where it has left off. Checkpoints can be used on machines of different type only if they are written in a hardware independent format. If this is the case, checkpoints are an ideal way to relocate an application in a heterogeneous environment. Hardware independence rules out data formats, which are based on memory images. Such memory images are e.g. used for checkpoints in the Condor High-Throughput-Computing environment [23], see Section 3.3.1.

Checkpoints can be drawn in regular intervals to safeguard the simulation against hardware failures. If the code dies for some reason, it can be restarted from the time on when the last checkpoint was generated.

Application Intelligence: A production code on a Grid is faced with a dynamic environment, which leaves the user in the dark on most of its properties during the execution of the program. The application needs to be skilled enough, to deal with these changing conditions. To give an idea, in which fields “application intelligence” is needed, a few examples are given:

- Memory Consumption: a simulation's memory allocation can depend on the behavior of the numerical problem. Memory consumption increases as regions of highly dynamic behavior require finer resolution (see section 2.2.3 on adaptive meshes). A code which is not aware of the memory constraints imposed by the hardware or queue settings usually terminates catastrophically.

- Runtime and IO awareness: Checkpointing is the process of saving the state of an application to disk. Depending on the size of the simulation memory and IO capabilities of the system, the writing of a checkpoint file can take considerable time. Scientists usually derive the correct duration of a checkpointing process from experiment and educated guesses. When users submit their simulations to a queuing system, the application needs to finish the writing of checkpoints before the queue time limit expires or the application is killed. In a Grid environment, where each resource has different I/O characteristics, checkpoint timing information is difficult to predict.

This list provides compelling incentives to introduce application intelligence: For example, an application which is aware that it is about to consume all available memory can either shutdown cleanly, stop any further refinement process or initiate a migration to a better suited machine.

2.3.5 Discussion – Probabilistic Reliability

Network, hardware and middleware are are the key components that turn the Grid into a productive entity for an application. Their characteristics are diverse, their properties fluctuate on all time scales and all components are unreliable. The key question is the following: *Is it possible to use the unreliable software, hardware and network components to create a reliable service structure ?* We believe that this can be achieved through the concept of *probabilistic reliability*.

This idea takes advantage of two abstractions: the abstraction of the application from the hardware and the abstraction of a services functionality from the application that provides it, also called "service instance". With this strategy we can install a redundant set of services: a single service type with multiple service instances. Multiple services types are orchestrated to form complex, compound services which also operate according to the same redundancy philosophy.

By deploying numerous services of the same type across a collection of machines, we create a pool of redundant service applications. While we cannot guarantee that a service on a specific host will be available at some time in the future, we can assume that the service type will be available *somewhere* on the collection of machines.

2.4 Implementing Scenarios – Thesis Objectives

The objective of this thesis is the development of a fault-tolerant service infrastructure that allows a client to operate autonomically. In particular, we permit clients to request high-level services like migration and spawn services through remote procedure calls from a server.

When we implement the advanced scenarios of Section 2.2, we are faced with two essential problems which have their origin in the characteristics of Grids:

1. The network and hardware components of the Grid, which host the various Grid applications and services, operate unreliably. However, we need to provide a service infrastructure, which yields a consistent access to services and resources.

2. The landscape of the Grid consists of a variety of middleware and software packages. Interfacing with these ubiquitous, legacy systems is critical for the success of a service infrastructure: we cannot promote a single system and require all sites to install it. Our service infrastructure needs to interface with today's software installations as well as upcoming technologies like web service based access methods.

The objective of this thesis requires multiple steps before a service infrastructure can be put into practice:

- We propose a consistent service environment by taking advantage of a probabilistic reliability through a redundant deployment of services. We develop this concept in Chapter 4 and call it in analogy to the Peer-To-Peer model "Grid Peer Services" (GPS).

- We establish a generic description model for objects on the Grid (Chapter 5). The "Grid Object Description Language" (GODsL) integrates the various independent aspects of Grid objects into a common information model. It allows for compact communication between Grid peer services.

- The GODsL data model is used to translate between the proprietary description vocabularies of the various Grid middleware solutions and it is used to interface with legacy applications, like batch submission systems.

- We develop a request handler framework in Chapter 6, to experiment with fault tolerance and error back-propagation capabilities in a distributed service environment.

- We propose an Application Information Server (AIS) as a registry to store *generic* information on services, files and resources, as well as client specific data (Chapter 8).

- We implement a number of fundamental GPS applications that provide basic Grid operations for clients through remote procedure calls (Chapter 7). These fundamental services use existing Grid infrastructure wherever possible.

- Based on the fundamental services we introduce a second GPS class that provides high-level services like migration and spawning (Chapter 8).

- By combining fundamental and high-level peer services, we demonstrate an implementation of a service monitor that is capable of managing redundant services: it is monitoring and restarting service applications as well as automatically recovering client simulations (Chapter 8).

- We apply the migration and spawn environment as well as the auto-recovery to two realistic scientific applications and analyze their autonomic migration and spawn behavior in Chapter 9.

Note that this thesis does not focus on the application internal realization of spawning. We provide a service environment to codes, which allows them to autonomically request such operations, while the service environment transparently performs the necessary tasks. Figure 2.4 sketches the distributed service infrastructure that is developed in this thesis. Hardware resources come in different types and constellations, they may be directly accessible or hidden behind firewalls. In an application-centric environment, these services are to be used by client codes without human interaction. The sketch shows several Grid Migration Servers (GMS), Application Information Servers (AIS) and Grid File Servers (GFS). The AIS functions as a central repository for information related to services, files or resources, etc. The GFS offers an access to file transfers for the participating machines and the GMS

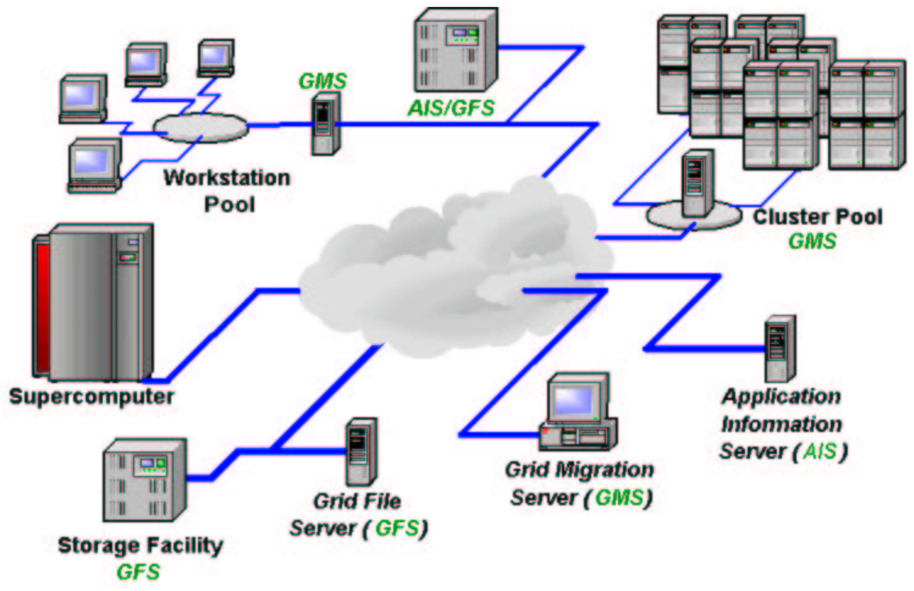


Figure 2.4: This thesis develops an information model and a fault-tolerant, distributed service infrastructure on a heterogeneous Grid: For instance, Grid File Services (GFS) use existing middleware to manage files and Grid Migration Services (GMS) offer migration services to applications. The AIS acts as a data warehouse for all participants and stores application, service, file and resource related information.

provides migration services. Services use existing Grid infrastructure and are deployed redundantly to compensate for hardware or network failure.

We will prove the working concept of the service infrastructure in Chapter 9, where we analyze the migration and spawn experiments that we conducted with realistic, scientific simulation codes on a testbed of machines.

Chapter 3

Grid Computing Environments and Related Work

This chapter acquaints the reader with different computing environments and middleware technologies for computational Grids. We want to raise awareness for the numerous solutions, each with an individual usage. We highlight some shortcomings of existing packages, which motivated our development of the *Grid Peer Services* and the *Grid Object Description Language* as the foundation for a distributed, fault-tolerant migration environment.

We start with an overview on resource management and application monitoring software in Sections 3.1 and 3.2, followed in Section 3.3 by a selection of high-level packages for seamless integration of resources and applications. Wherever possible we will use these existing Grid solutions in our migration environment. We introduce the Cactus Code Framework in Section 3.4 as an application based Grid solution. We take a look at information models to describe Grid entities in Section 3.5 and list some of the major Grid research initiatives in Section 3.6. We conclude with a comparison of the introduced software in Section 3.7. The web service model and related technologies, like OGSA are introduced in Chapter 4 and discussed in Section 5.11 together with GODsL.

3.1 Resource Management and Monitoring

In this section we give an overview on resource and job management systems which are relevant for our work. An automated submission service as employed in our migration service environment has to interface with a large variety of resource management systems.

3.1.1 The Globus Toolkit

The Globus Toolkit provides solutions for a variety of tasks in a computational Grid. Globus offers central services for communication, security, information management, brokering and resource access. The modular approach in Globus is contrasted by the Legion framework [31], in which every component of the Grid becomes an object of a single virtual computer [52]. In this overview, we focus on the Globus resource management, information infrastructure and security packages. Like all Grid enabling middleware, Globus is a software layer that is located above the operating system and below the site specific management applications, such as batch submission systems and user applications.

Resource Specification Language

The Globus Resource Specification Language (RSL) provides a common interchange language to describe resources requirements, based on attribute-value pairs. The various components of the Globus Resource Management architecture use RSL strings to perform their management functions in cooperation with the other components in the system. The RSL allows for complex resource descriptions. (An RSL script, generated by the migration service for a three machine meta-computing run is shown in Section 7.4.2). Below is a simple RSL script for a parallel run:

```
(&(resourceManagerContact="fermat.cfs.ac.uk/jobmanager-pbs")
   (count=16)
   (jobtype=mpi)
   (executable=pi)
)
```

Such scripts can be launched with Globus, as shown below:

```
lanfer@vidar2:˜> globusrun -f RSLfile
```

The attributes (e.g. count) are mapped to the corresponding batch submission attributes by the Globus Resource Allocation Manager (GRAM) [49]. Here the job is submitted through PBS. The jobmanager is omitted for interactive execution. The set of RSL attributes is fixed[1].

Metacomputing Directory Service

The Metacomputing Directory Service (MDS)[2] is part of the Globus information infrastructure and provides a directory service for Grids. MDS is a framework for managing static and dynamic information about the status of a computational Grid compute nodes and storage systems. MDS uses the Lightweight Directory Access Protocol (LDAP) [89], as a uniform interface. MDS consists of an information provider component called “Grid Resource Information Service” (GRIS), which is deployed at participating sites, and a centralized directory component called “Grid Index Information Service” (GIIS). An MDS can be queried through a graphical user interfaces or an API. The effortless browsing of a MDS database through a GUI is deceptive: searching and accessing specific data items in deeply rooted MDS tree from within an application is non-trivial.

Globus Security Infrastructure

The Globus Security Infrastructure (GSI) allows for secure authentication based on X.509 public key certificates. GSI can be used to authenticate users, resources and processes. A request to access a machine is authorized if a user has an entry in the machine’s “grid-map-file”, which serves as an access control list. GSI operates with global user IDs (see `Your identity:`), which are mapped to the site local accounts. The grid-map-file must be edited on all machines each time a user is added to the Grid environment.

Single sign-on is accomplished through the generation of a temporary proxy [87] from a user-certificate, as shown below. The proxy can then authenticate the user to other machines or processes. A proxy has certain lifetime, which defaults to 12 hours. During this time, the user can access machines, which have the user’s identity in their grid-map-file. After the proxy expires a new one must be retrieved manually. A proxy can be taken along when users log into other sites. This is called “proxy-delegation” and allows a user to access all participating machines from any other.

```
origin> grid-proxy-init
Your identity: /O=Grid/O=Globus/OU=aei.mpg.de/CN=Gerd Lanfermann
Enter GRID pass phrase for this identity: ********
Creating proxy ......................................... Done
Your proxy is valid until Tue Aug  6 00:13:56 2002

origin> grid-proxy-info
subject  : /O=Grid/O=Globus/OU=aei.mpg.de/CN=Gerd Lanfermann/CN=proxy
issuer   : /O=Grid/O=Globus/OU=aei.mpg.de/CN=Gerd Lanfermann
type     : full
strength : 512 bits
timeleft : 11:56:59
```

[1]http://www-fp.globus.org/gram/rsl_spec1.html

[2]MDS was recently renamed to “Monitoring and Discovery Service”. The literature uses both terms.

3.1.2 Condor Classified Advertisement

While we discuss the full Condor environment in Section 3.3.1, we focus here on a component of Condor, called the Condor Classified Advertisements (Class-Ads) [22]. Class-Ads use a semi-structured data model, which folds the query language into the data model, allowing applications and computers to publish queries as attributes. Class-Ads are exchanged by Condor processes to schedule jobs. They include a matchmaking ability which is a valuable tool to compare resource requirements and constraints.

Below, we show two Class-Ad examples: the left one describes the characteristics of an object (an apartment), the right one describes the requirement of a requestor (apartment renter).

```
[                                    [
  MyType = "Apartment"                 MyType    = "ApartmentRenter"
  SquareArea = 3500;                   Student   = True;
  RentOffer  = 1000;                   Rank      = 1 / (other.RentOffer)
  OnBusLine  = True;                   Constraint= other.BusLine && otherSquareArea>2700
]                                    ]
```

The arithmetic and relation operator can be part of a Class-Ad. They let the potential renter specify constraints for all objects (they must be near a bus line and have be larger than 2700 sft.) and rank them (cheapest offer first). It is obvious how this mechanism can be perfectly applied to the evaluation of resource and network constraints. We will utilize this package to match application requirements with resource information that we have gathered from various information services.

3.1.3 Batch Submission Systems

Batch submission systems provide a mechanism for submitting, launching and tracking jobs on machines. These systems greatly simplify the use of clustered resources. They achieve an optimal utilization of the system and provide a quick turn around for the user. The commonly known systems are the *Portable Batch Submission System (PBS)* [54], *Load Leveler (LL)* [58]. The *Load Sharing Facility (LSF)* [92] and *Sun Grid Engine (SGE)* [83] also provide batch submission functionality, but their capabilities reach further and are discussed in Section 3.3.

Traditionally jobs are submitted through user written scripts. A sample LSF script is given below. A script specifies the resource requirements of the application through directives (`#BSUB`). The commands to be executed trail the list of directives. In the example, a job requests 84 processors and 10 GByte of memory. The executable is launched through `mpirun`.

```
#!/bin/sh
#BSUB -M 10G
#BSUB -n 84
cd /utmp/gerdlan/AHF
mpirun -np 84 ./cactus_ahf GravWave2.par
```

The Maui Scheduler: Maui is an application scheduler and designed as a policy engine to organize when, where and how compute resources are allocated to jobs. The Maui scheduler receives information on the resource and the scheduled application from a resource manager like LoadLeveler, PBS or LSF. If a new parallel job is to be scheduled, Maui calculates the optimal location of the job in terms of node location and execution order. Maui supports *advanced reservation*, which reserves a block of compute time for a user. The scheduler arranges competing jobs in a manner that the reserved resources are available by the time the reservation starts.

3.2 Application Monitoring

In this section we review a number of monitoring systems for applications, which allow a program to extract information *at runtime* on its own performance or its environment, e.g. network situation. Such monitoring tools are pre-requisite for autonomic operation.

3.2.1 PAPI, SvPablo and PACE

The Performance API (PAPI) [65] provides a uniform API for accessing hardware counters on microprocessors. SvPablo [78] is a language independent performance analysis and visualization system that supports analysis of applications executing on both sequential and parallel systems. In addition to capturing application data via software instrumentation, SvPablo also exploits hardware performance counters to capture the interaction of software and hardware. Both hardware and software performance data are summarized during program execution. Performance predictions is an important field of research that attempts to qualify and extrapolate performance data into the future. PACE [71] is such an attempt for parallel applications. Ripanau [77] targets in his work the predictability of algorithms.

3.2.2 Network Weather Service

The Network Weather Service (NWS) [91] monitors the network quality between sites and makes this information available to applications. It also attempts to predict the TCP/IP end-to-end throughput and latency that is attainable by an application. Pchar [67] is another tool to measure the bandwidth and latency. We will use NWS and related technology in a future project (see Section 5.8.3) to rank hosts based on their accessibility for large file transfers.

3.3 High-Level Grid Environments

In this section we list related high-level environments, which integrate several capabilities into a single system (like resource monitoring, application submission and migration capabilities).

3.3.1 Condor

Condor [17] is a high-throughput scheduling mechanism, which is developed at the University of Wisconsin, Madison. Condor has a successful ten year history in high-throughput computing. Condor allows users to submit jobs to machines of an administrative domain called “flock” and harness them in a “cycle-stealing” mode. Besides scheduling jobs, Condor suspends or relocates an application if the computer is used interactively, or if the machine load increases above a certain threshold. Within a homogeneous environment single-processor jobs can be checkpointed and restarted by the transfer of the application’s memory image. The connection of multiple administrative domains is the focus of Condor-G [40], which is a project to join multiple condor “flocks” from independent organizations through Globus. Our migration environment can for example interface with Condor to fan out single processor spawn jobs.

3.3.2 Sun Grid Engine

Sun Grid Engine (SGE) [83] and the enterprise edition (SGE-EE) offer a distributed computing environment for Grids within a single organization. SGE define complex rules for resource sharing: It features guaranteed compute capacities and has a “deadline”policy to dedicate compute capacities for the duration of a project. SGE distinguishes three types of resources (CPU cycles, Memory, and I/O activity) to calculate the user's share of the available resources. Sun has integrated its solution into a single product.

3.3.3 Load Sharing Facility

Load Sharing Facility (LSF) [92] by Platform Computing is a distributed computing environment, similar to the Sun Grid Engine. In its base configuration, LSF offers a batch submission system similar to PBS. The LSF base product can (or must) be complemented with additional packages to gain advanced resource management capabilities.

3.3.4 TENT and Symphony

TENT [35] is a distributed workflow management system for engineering applications. It originated in the field of aircraft and turbine design, where different commercial and non-commercial software packages are used sequentially in the process of data preparation, simulation and analysis. TENT uses CORBA [24] in its communication middleware and Java for the key components. The TENT framework itself is a monolithic solution, designed for local Grids. A Globus based solution for wide area networks is currently researched.

Symphony is a component based framework for composing, saving, sharing and executing meta-programs [80, 66]. The Symphony framework abstracts Grid architectures and their middleware. Remote programs and files can be accessed using a number of different protocols and services: job submission is achieved through GRAM and through a proprietary protocol, file access is offered through HTTP, FTP, GSI-ftp.

3.3.5 Harness, Javelin and Charlotte

Harness [69] (Heterogeneous Adaptable Reconfigurable Network System) is an experimental meta-computing system that is based on the concept of a distributed, virtual machine (DVM). Harness aims to overcome PVM limitations, e.g. restricted communication scope between PVM virtual machine. It allows multiple PVMs to operate together.

Javelin [70] connects multiple, anonymous machines through Web java-applications. Users download a client as a java applet and participate instantaneously in ongoing computations. The system is aimed at trivially parallel applications. A broker process manages clients and distributes the work.

Charlotte [16] is a virtual machine that executes Java applications through Web browsers. Charlotte provides load balancing by re-assigning tasks to faster machines. Like Javelin, it targets trivially parallel computations.

3.4 The Cactus Code Framework

The Cactus Code framework [47] addresses a number of characteristics in a Grid environment on the client side, rather than in a Grid framework. We give a brief overview on this project and its philosophy. We use Cactus Code based simulations as clients for our migration experiments and it serves as a the framework for our service applications.

The Cactus Code is developed at the Max Planck Institute for Gravitational Physics by both physicists and computer scientists[3]. The Cactus Code embodies a new paradigm for the development of numerical software in a collaborative and portable environment. As a freely available open-source

[3]The author is a founding member of the Cactus team.

toolkit, Cactus extends the traditional, single-processor code development into parallel applications that can be run on a large variety of platforms, from laptops or clusters of workstations, up to supercomputers. Cactus provides access to advanced computational tools, such as parallel I/O, remote visualization and steering, as well as performance monitoring [6].

The name “Cactus” comes from the design principle of a module set, termed “thorns” that interface with the Cactus framework, called “flesh”. In emerging research areas, it is not always clear what techniques are best suited to solve a scientific problem. Research groups in large scale scientific challenges are usually distributed and require a collaborative work style that needs to be mirrored in their programming environment as well. Further, the underlying computing platforms are evolving and changing rapidly over time. For these three key problems, the Cactus Code framework offers a solution for non-computing-experts like numerical scientists: The modular design allows scientists to assemble an application from a set of numerical thorns and tune it to solve a certain problem. Thorns can be replaced to try out different algorithms or they can be added to perform additional analysis on the evolved data.

The Cactus flesh controls how thorns work together by coordinating the data and process flow. “Driver thorns” abstract the parallelization, which allows a Cactus application to replace e.g. MPI with other communication libraries like Globus-MPI or PVM. The Cactus Code has thorns that interface with the PAPI and SvPablo application monitoring tools introduced above. Cactus executables can be compiled for numerous architectures. All these features makes Cactus an ideal framework to experiment with autonomic applications in a Grid computing environment [11, 7, 5].

Scientists at the Max-Planck-Institute use the Cactus Code for developing numerical solutions to Einstein’s equation. These computationally intensive partial-differential equations describe cosmic events like the collision of black holes and are solved on large-scale supercomputers [12].

In this thesis we use the thorn concept for our Grid Peer Service implementation to attach an arbitrary number of service thorns to a service request handler, described in Chapter 6. This approach allows us to generate executables whose service variety can be fine-tuned to a specific task.

3.5 Object Description

This section takes a brief look at data models that are used to characterize complex objects on a Grid, going beyond isolated aspects such as resource descriptions. The task of defining objects in a simple, robust way is not new. In fact, all of the high-level Grid environments must have some data model which describes objects on a Grid more or less coherently. However, most of these data models are internal and cannot be used to communicate information externally between services.

3.5.1 Symphony

Symphony provides Java beans to describe file and program properties. A Java file bean can e.g. represent local or remote files. The beans act as “data sinks” or “data sources” [80], which read or write files, respectively. File beans include file description and transport. A Java program bean is a description of a local or remote process. Beans can be coupled together: for instance two Java file beans imply source and target file in a copy operation. We take a similar approach with Grid Objects, which express source and target object in a copy service request.

3.5.2 GrADSoft

The GrADSoft (Grid Application Development Software) environment seeks to provide a continuous program preparation and execution system, utilizing MDS and NWS. GrADSoft uses those native data models and as well as its own *Abstract Resource Topology Model (AART)*. The AART bundles resource, network and topological information into a *virtual machine* which is interpreted by the GrADSoft scheduler.

3.5.3 CIM and CIM Application Management Model

The Common Information Model (CIM) [21] is developed by the Distributed Management Task Force (DMTF). CIM's data model is an implementation-neutral scheme for describing overall management information in a network/enterprise environment. CIM is comprised of a specification and a scheme: the specification defines constructs of the Managed Object Format (MOF) language, while the scheme provides the actual model descriptions. The CIM Application Management Model is a CIM based model to describe the details commonly required to manage software products and applications. It distinguishes an application into software products, features and elements. The CIM model uses three concepts to structure an application: The *software element life cycle* describes activities like deploying software; the *environmental conditions* list dependencies for an applications (e.g. existence of certain directories); *software element actions* describe the sequence of actions that lead to the generation of a new software element.

CIM has similarities regarding our goal to provide a common description system. CIM and CIM-based information models offer a functionality which reaches far beyond what we feel is necessary to describe entities on the Grid. We follow a more simplistic approach to describe Grid Objects. However, CIM development highlights the need to have common, abstract data models to manage complex interactions.

3.6 Grid Research Initiatives

We list some of the major multi-organizational research initiatives, that contribute relevant Grid infrastructure.

- **GrADS:** The GrADS [48] project is developing strategies for launching programs in Grids and developing GrADSoft (see Section 3.5.2). GrADS maintains a testbed and studies Grid computing scenarios.

- **GridLab:** The GridLab[82, 51] project focuses on developing capabilities to use dynamic resources. It develops service tools to make applications aware of their computing environment. The migration environment developed in this thesis serves GridLab as a prototype to study the behavior and requirements of migration applications in a Grid.

- **Global Grid Forum:** the Global Grid Forum (GGF) [43] is a forum of researchers and practitioners working on technologies. GGF focuses on the promotion and development of Grid technologies and applications. The GGF has inaugurated a Peer-To-Peer working group in September 2002[4], which investigates how to use OGSA (see sec. 4.5.2) with P2P environments.

- **EGrid:** The European Grid Forum has merged its organizational structure with the GGF. The EGrid still maintains a testbed to test out Grid technologies. We are using this testbed for our migration experiments.

[4]`http://batalion.ucsd.edu/ggf`

3.7 Discussion

In this section we evaluate the presented Grid software with regard to *fault-tolerance* and *interoperability* as well as their *application restrictions* and the *type of Grids* they are best suited for.

The above frameworks manage applications on the Grid, in the sense that they submit jobs and copy files. But they do not support autonomic applications in the sense that a self-contained application requests services or contributes information to a shared database. The situation is slightly ironic: On one hand, applications are made increasingly intelligent by packages like by SvPablo [78] or PAPI, on the other hand the applications are denied an environment which executes their decisions, for example the conclusion to migrate if an SvPablo probe indicates inadequate resource utilization.

Many information registries are designed to store specific data: e.g. network data (NWS) or machine information (MDS). We will see later in Chapter 5 that this is an inefficient abbreviation of a comprehensive situation: e.g. the network bandwidth to a site and its disk capacities define constraints for large file transfers and must be expressed together. Further, autonomic, self-aware applications become a source of information in its own, when they benchmark their host's I/O system or floating point operations. To tap into this information pool, registries must permit data retrieval *and* deposit of flexible content by user applications. This view is not found in the traditional design of information registries.

TENT and many other not mentioned packages rely on their own server components in a distributed environment. This is a strong requirement for a global Grid, in which each site has administration autonomy. Harness, Charlotte and Javelin need Java applications to operate in a heterogeneous Grid, underlining the fact that the potential of heterogeneous Grids is still not fully explored for real-world applications: middleware designers compromise with respect to the language (Java) or require a homogeneous machines pool.

While most systems do pay attention to the issues of security in a Grid environment, most systems do not sufficiently address the problems which are caused by hardware unreliability. For example, the Condor environment detects and restarts a failed Condor client, but it is not obvious how the master program is monitored. Symphony provides a small foot print framework that can be used on top of computational Grids without major modifications or requirements on the host system, but does not address fault tolerance either. SGE restricts its scope to “campus-Grids” right away. In general, the above solutions concentrate on handling small and enterprise-sized Grids. They rely on centralized services, which is acceptable for small sized machine pools. Due to unreliability of the Grid components these solutions do not scale up to *global* Grids. This is not a failure of the Grid middleware but caused by the unreliability of the Grid components.

It is also not clear why only so few of the high-level systems link up with related Grid infrastructure. We believe that interfacing with existing technology is crucial to cover a significant percentage of machines in a global Grid and to avoid the re-invention of the wheel. A main difference between existing environments and the Grid Peer Service model developed in this thesis, is that currently each system provides a solution within an isolated universe. This criticism is not valid for the Globus Toolkit and a few others, which not only support a variety of batch systems but also interface with related systems like Condor.

The Grid Peer Service model (GPS) should be seen as a prototype for providing a service environment, which is not encapsulated. It interfaces with existing middleware wherever possible to extend its services to as many machines as possible. Grid Peer Services do not need to run on every host, instead GPS instances interface through API's or interactively with the infrastructure on remote machines (MDS, PBS, etc.). In this spirit, Grid Peer Services do not intend to compete with any of the Grid middleware solutions mentioned above. Instead it should be seen as an approach to enhance the usability and interconnectivity of Grid middleware for autonomic applications.

Chapter 4

Grid Peer Services

The *Peer Service Model* and the *Web Service Model* emerged and evolved separately from each other and are two new trends in the field of distributed computing. Both enable users to participate in a deeper vision of the Internet. In this chapter we introduce the two models and explain how they can support a fault tolerant service structure on Grids. The two models are discussed in Sections 4.2 and 4.3, their fusion is analyzed in Section 4.4. Current web service technology is reviewed in Section 4.5, the common encoding and transport protocols are outlined in Section 4.6. The chapter concludes with a discussion of our service approach in Section 4.7.

4.1 A History of Services

Until recently developers and users did not have to be overly concerned about the hardware independence and communication skills of their software. In a "server-centric" environment the application clients had to know about the properties of the local hardware and software, since all functionality and resource considerations were made relative to a central server environment. There was no need to accommodate third party hardware or software outside the scope of the local server theater, simply because there was no information flow of significant size, e.g. between two cooperating companies. As the need for information interaction increased with the appearance of "business to business" (B2B) models, web services emerged as a concept which allowed two applications written in different languages, employing independent internal data structures and hosted on arbitrary hardware to communicate.

Other than the commonly known web interfaces, which are accessed through web browsers, web services are designed for the automatic communication *between applications*. Technologies like "Extended Markup Language" (XML)[18], "Simple Object Access Protocol" (SOAP)[28], "Web Service Discovery Language" (WSDL)[20], "Universal Description Discovery and Integration" (UDDI)[88] and most recently the "Open Grid Services Architecture" (OGSA)[37] enable applications to autonomously connect to other programs to exchange information and services.

Peer-To-Peer (P2P) computing has been made popular by applications like seti@home, Napster or Gnutella and opened a new view on how to anticipate the Internet. Returning to the roots of the Internet, applications and users interact directly with each other, without involving a Web server or a centralized administration unit. This chapter discusses why the fusion of web services and the peer-to-peer model is an ideal match for the anatomy of a computing Grid, as outlined in Section 2.3. We illustrate how an *environment of web services* can be created and how the P2P service model endows services with a redundancy that is able to handle the unreliable properties of computing Grids.

This service topology serves us later in the implementation of the Grid Peer Services in Chapter 7 and 8. The GPS implementation allows us to realize the different Grid scenarios that were outlined in Section 2.2. The experiences and experiments with the Grid Peer Service model are discussed in Chapter 9.

4.2 Web Service Model

The core idea of a web service is simple: web service technology decouples a service from the underlying hardware and software and makes it available to the outside community through a well defined interface. A web service consumer does not have to be concerned about the implementation of the service. This abstraction of the *service* from the actual *implementation* has a variety of advantages to both service providers and users: web service providers can upgrade their hardware without impact on their clients. Clients on the other hand are not forced to adapt the same IT growth rate as their service providers. Clients may choose from several service offers as long as the functional capability is identical. An application which is web service enabled and which has outsourced major portions of its functionality to external applications is more portable and leaner in design.

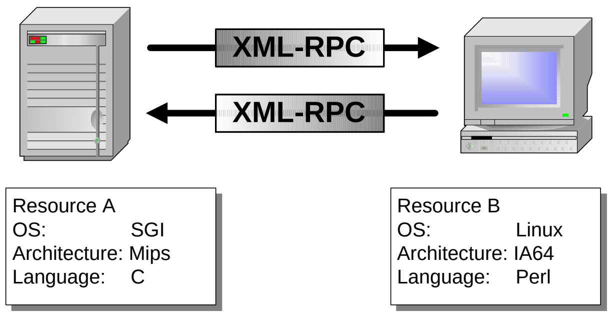


Figure 4.1: Web services hide details as architecture, programming language or internal data structures. Communication is done through protocols like XML-RPC or SOAP, which provide platform-neutral encoding through XML and use HTTP as for message transport. Arbitrary applications are able to communicate if they conform to these communication standards.

A web service can be seen as an interface positioned between two application codes. The web service acts as the middle layer, which allows the communication of the two codes as it hides program specific characteristics such as programming language, application internal data layout, architecture of the hosting machines, etc. as shown in Figure 4.1, where two applications on diverse hardware and software platforms communicate through a common protocol. Any application which speaks this protocol can access the functionality hidden behind the interface. Because web services are defining the language and transport of the *communication*, conforming to these standards is a strong requirement to interact with such a service. XML-RPC and SOAP are such a protocols and can be called a *lingua-franca* of web services. Any number of applications can join this ensemble of communicating tasks. The pool of participants is distinguished into "service providers" and "service requestors" in analogy to the client-server model.

The abstraction can be extendend to the point that it is not only irrelevant *how* a service is realized but also *where* it is operating. This abstraction requires a third party, which traces the types and location of services: It is termed *Service Directory* or *Service Registry*. We deduce that a web service environment (Figure 4.2) consists of three elements:

- The *Service Provider* is an application which has the ability to perform a certain task. It makes the task available to external programs. The service provider is contacted by a service request. It executes the request in accordance with policies like security, authorization or priority.

- The *Service Requestor* is an application which wants to use the functionality provided by a service. The requestor sends a message to the provider and requests a certain service.

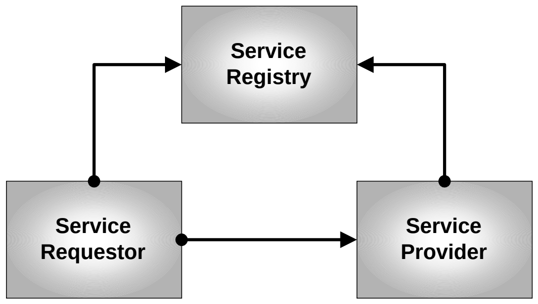


Figure 4.2: Web service components can be categorized into *Service Providers*, *Service Requestors* and *Service Directories*. Since the location of a service provider is not necessarily known to a service requestor, a directory service is needed to relay service location and contact information.

- The *Service Registry* stores information on the available services and their locations. The registry is contacted by the provider, who *announces* its service and contact information. The registry is *queried* by the service requestor to obtain the location of a services.

Web Services vs. Client-Server Operation: The term "web service" and "client-server" are often used synonymously. The understanding of web services is more restrictive than what is usually described by "client-server" computing. While every web service can be regarded as a temporary client-server relationship, not every client-server pair is a web service. Web services require the abstraction of functionality, implementation and architecture.

4.3 Peer-To-Peer Service Model

Peer-to-Peer (P2P) computing has been the result of a trend towards decentralizing software services and hardware, away from monolithic client-server systems to distributed environments. The peer service model has many features which are also found in the web service model. P2P services do not need to conform to the web service definition. A typical feature of a P2P environment is the vast quantity of participants compared to the number of services. This strategy is not captured by the web service model. It endows a P2P environment with a *probabilistic* immunization against the unreliability of a single peer: For a large enough number of service providers in a P2P pool, a service has a good chance of being available "somewhere". P2P networks can "survive" in environments which are too unstable for a single consumer-requestor service relationship.

Peer-to-Peer style computing yields a probabilistic (rather than deterministic) strategy for service reliably. P2P shifts the focus toward collaboration and communication oriented applications, while the web service model is more aimed at describing the communication in a temporary client-server relationship.

4.3.1 A P2P Example: Gnutella

One of the widely-known P2P networks is built by Gnutella [44]. Gnutella peers use point-to-point communication to connect with neighboring peers. Gnutella is used as a file sharing utility that enables hundreds of thousands of users to offer, search and download files over the internet, including mp3-encoded music files. In order to locate a file, a peer sends a request to its neighbors, which propagate this message on through the Gnutella network. If a peer has the requested file available, it returns a response. Each request has a time-to-live which is decremented at each peer hop to avoid the problem of network flooding. Gnutella's communication behavior has been the focus of research by Ripeanu [77]. File sharing relies on the redundancy of service and data: a failing peer does not effect the functionality of the peer network nor the availability of a certain file, provided the network is large and information (files) are redundant.

4.3.2 A P2P Characterization

Based on the example above, we can derive a classification of a Peer-to-Peer system, since not every distributed application is P2P:

- P2P consists of a number of peers, each performing a specific role in the P2P theater.
- The total number of peers is large compared to the numbers of roles they play.
- No constant connection between two individual peer application is required to provide a service.
- The client-server classification is dissolved, peers inter-operate on equal terms.
- Peers agree on a common transport protocol.

The three fundamental components of the web services are also present in the Peer Service Model. A key problem imposed by a P2P environment is the coordination and monitoring of the independent participants. The number of peers that a service registry has to deal with is significantly larger. The web service model emphasizes the client-server classification, while the peer service model stresses the fact that a service requestor and a service provider operate "at equal terms": a peer can both act as server and as a client. Furthermore, P2P establishes a data and operation redundancy and outnumbers possible failure points with pure quantity.

4.3.3 Why P2P ?

We saw that the Peer-To-Peer service model has a number of advantages which are not found in the web services model.

- P2P services are usually small, simple and flexible. Ideally, they scale across a vast number of hosts, which yields a probabilistic redundancy for a service type.
- P2P eliminates the single application bottleneck since services and data are provided by more than one application.
- P2P provides easy load balancing. P2P service application can be shutdown or recreated in the case of high service demand.

Many of these characteristics are also possible in non-P2P environments. For example, the threading of tasks is a standard procedure in server computing. The web service idea goes further and adds the notion of application independence: the service interface becomes the essential component, not the application itself.

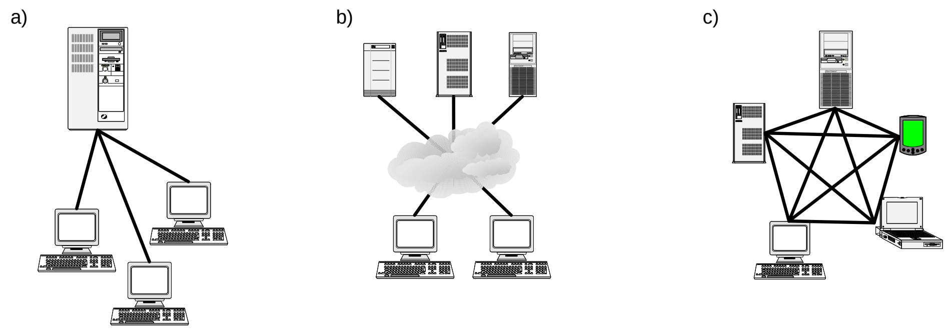


Figure 4.3: Communication patters developed over the years from the isolated client-server model (*a*) to a web service model (*b*), which allows various clients to communicate content with servers of different domains and architectures. In the most recent – and still evolving – step (c), the distinction of client and server is dissolved as participants in peer network communicate on equal terms.

Extending the P2P idea: The principle of P2P abstraction can be applied to other situations as well, for instance:

1. A compute resource conforms to the idea of a P2P network, when compute capacity of a computer is decoupled from its specific plattform and location. In an (admittedly) ideal world, an application is requiring a certain compute power and is not be restricted by the hardware, which delivers this capacity. The analogy to web services is evident.

2. A number of different services are combined to provide a complex “compound service”. The compound service takes advantage of the redundancy of the lower-level services and can be deployed redundantly as well. This construction is e.g. used for the Grid Migration Server in Chapter 8.

4.4 The Concept of Grid Peer Services

It is obvious, how web services and a P2P strategy complement each other. Web services bring the abstraction of services from the underlying hardware and implementation. P2P enhances this model with a reliability due to its service and data redundancy. Figure 4.3 sketches the evolution of the communication patterns among applications: to the left a traditional client-server model is shown, which operates in an encapsulated environment. Web services (shown in the center broke) this crust and allowed multiple clients to access services on various hosts of different administrative domains. To the right, the next – and still emerging – step is shown, which opens up the client-server distinction and allows participants to communicate on equal terms.

The Grid Peer Service models combines both models as complements the single edged web service model with a peer to peer topology. We define Grid Peer Services as follows:
Grid Peer Services are a service environment, in which each particular service is realized through multiple service instances, which operate independently of a hardware or software in multi-organizational Grids and communicate through web service interfaces.

The P2P model supplies a robustness against hardware failures to the web service model by giving a particular service a high redundancy: the service is likely to be available “somewhere” even if some machines or networks fail. We are interested in providing exactly this fault-tolerant property to our migration services.

4.5 Web Service Technology

In this section we give an overview over existing web service technology. We introduce the standards of WSDL, UDDI and WSFL in Section 4.5.1 and review OGSA as an architecture aimed at Grid environments in Section 4.5.2.

4.5.1 WSDL, UDDI and WSFL

The section reviews the web service technology to describe, identify and concatenate web services.

Web Service Discovery Language (WSDL): WSDL [20] is an XML based language, which defines the interface of a service. WSDL informs an application, what kind of interface specification a service provider is using. WSDL is similar to “Interactive Data Language” (IDL), which is used to characterize CORBA interfaces. Service interfaces are defined abstractly in terms of message structures and sequences of simple message exchanges, called “operations” in WSDL terminology. The interface description is tied to a concrete transport protocol and data encoding scheme. WSDL is extensible and allows the complex interface definitions for various network interfaces and transport procedures. Several standardized binding conventions are defined, describing how to use WSDL in conjunction with SOAP, HTTP and MIME.

Universal Detection and Discovery Interface (UDDI): UDDI [88] operates on the level of service registration and service discovery. UDDI stores the description of services while WSDL describes the service itself. UDDI allows a business to register information about the web services they offer so that other businesses can find them. The core component of UDDI is a XML based “business registration”, which consists of “white pages” including address and contact identifiers, “yellow pages”, including categorizations based on standards and “green pages” containing technical information about the service that are provided by a business.

Web Service Flow Language (WSFL): WSFL [64] by IBM is an XML language that models the composition of web services. It specificies the execution sequence of the functionality provided by the composed web services. Complex services are specified by defining the flow of control and data between web services. In August 2002, the ideas of IBM’s WSFL and Microsoft’s XLANG were converged into the *Business Model Execution Language for Web Services (BPEL4WS)*[1]. BPEL4WS defines an interoperable integration model that aims at facilitating the expansion of automated process integration in intra-corporate and business-to-business environments. We do not use WSFL nor BPEL, but suggest in Section 8.4.5 the dynamic definition of high-level services like the introduced migration service through a service flow definition.

4.5.2 Open Grid Services Architecture

The Open Grid Services Architecture (OGSA) [37, 86] is an evolution of the current Globus Toolkit [36] toward a Grid system architecture. It is based on the integration of the Grid properties and the Web service strategies and technologies. An initial set of technical specifications composed by the Globus

[1]http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

Project and IBM has been proposed at the Global Grid Forum in February 2002. This specification is currently open to input from the community, and is discussed within the Global Grid Forum.

The Open Grid Services Architecture is the most recent approach to combine the Web service properties such as service descriptions with features like dynamic service creation. OGSA is likely to have a significant impact on the web service idea. The Grid Service Specification[2] states that OGSA is intended to combine key Grid technologies and web services into system framework based around the Grid service. The OGSA specification defines with WSDL extensions the interfaces and mechanism that are required for creating distributed systems. It includes features like "lifetime management" for services and service "notification". The OGSA specification addresses authentication, authorization and reliable startup of services. Similar to the Grid Peer Services (GPS) system in Chapter 7, OGSA also allows the construction of services through the composition of multiple lower-level services.

A first analysis of the OGSA specification has been given by Gannon et.al. [42]. We cannot provide an indepth description of OGSA in this thesis, but put the objectives of OGSA in relation to our work in Section 5.11, after we have introduced the Grid Object Description Language (GODsL).

Web Service Security: Traditionally, web services are served from a web server. Because web service requests masquerade themselves as web traffic, they usually pass right through firewalls via web port 80. This does pose a security risk, since it allows access to functionality inside a machine. In a secure environment it mandates that the web service supplies a security system of equal strength as the firewall concept.

Web services which are generated by users, like the GPS applications, execute with usermode permissions. We use ports other than 80 and above the restricted port range of 2000. Numerous solutions exist that address the issues of security and authentication during and after request transport, like the reliable HTTP (HTTP-R) by IBM. Because we wanted to develop an innovative service environment for autonomic applications, we have not made the security aspect a major theme in this work.

4.6 Web Service Encoding and Transport

Information which is exchanged between programs needs to be packaged into a format that is understood on both sides. Common packaging formats for web services include SOAP (Section 4.6.2) and XML-RPC (discussed in Section 4.6.1), which is an early implementation of the SOAP standard. Both SOAP and XML-RPC are XML based encoding methods. The process of converting application internal data into a common XML format is called *serialization*. The inverse process of extracting data and feeding it back into application internal structures is called *deserialization*. XML is a meta-language which allows the expression of complex data types and structures, while keeping them easily traversable. XML is vendor and platform independent, as well as language and transportation neutral.

Moving serialized data over the wire is described by transfer protocols. It is important to note that data transport is independent of the encoding. Web services may be built on top of nearly any transport protocol. In a web service model, participating applications have to agree on the encoding and the transport.

The most common transport protocols of web services are network protocols such as "Hypertext Transfer Protocol" (HTTP) [34], "Simple Mail Transfer Protocol"(SMTP) [76] or "File Transfer Protocol" (FTP) [75]. The transmitted content in a web service is independent of the chosen transport layer. This "transport-neutral" property allows one to change the underlying transport protocol without modifications to the service implementation. In this thesis, we use the most commonly employed transport protocol: HTTP.

[2]http://www.globus.org/ogsa

4.6.1 XML-RPC

XML-RPC (“Extended Markup Languages - Remote Procedure Call”) [90] provides a XML-based mechanism for making function calls across a network. It emerged in early 1998 as a spin-off of the SOAP protocol development and was published by Userland Software[3].

XML-RPC allows an application to specify a method name and a number of arguments as part of a request. The request is sent to a server, which deserializes the message, makes the appropriate local function call and passes on the transmitted arguments. The response of the function call, which can be anything from an error code to a database entry is serialized to an XML-RPC document and returned to the requestor (Figure 4.1 on page 26). The XML-RPC vocabulary consists of simple data types and structures. XML-RPC has no notion of objects nor mechanisms for providing translations to other XML vocabularies. Despite these limitation, XML-RPC has proven to be a robust protocol and it successfully used for numerous projects.

Data Types: XML-RPC provides the following data types, listed here with a brief example:

Data Type	**Example**
integer	`<i4>42</i4>`
Double	`<double> 3.1415 </double>`
Boolean	`<boolean>0</boolean>`
String	`<name> North Dakota </name>`
dataTime.iso8601	`<dateTime.iso8601>` `19040101T05:24:54` `</dateTime.iso8601>`
base64	`<base64>R0lGODlhFg</base64>`

Data Structures: XML-RPC supports the two kinds of data structures, which can be mixed and nested:

- **Structures** identify a value with a string-typed key. Structures can be nested: the value tags can enclose sub-substructures or arrays.

```
<struct>
  <member>
    <name>Key</name>
    <value>Value</value>
  </member>
  <member>
    <name>Key</name>
    <value>Value</value>
  </member>
</struct>
```

- **Arrays** are a list of values. The values do not need to be of homogeneous type. Within the value tags, any of the above described data types are allowed. Arrays can also contain sub-arrays and structures.

```
<array>
  <data>
    <value>... </value>
    <value>... </value>
  </data>
</array>
```

[3]The XML-RPC specification can be found at `http://www.xmlrpc.com/specs`

Request Structure: Each request in an XML-RPC consists of a single XML document, whose root element is marked by `<methodCall>` and closed with `</methodCall>`. The body of the method call is tagged with `<methodName>` (`</methodName>`), and names the function which is to be executed. The tag `<params>` (`</params>`) encloses the list of parameters and values, which is supplied as an argument to the requested method.

Below is a sample XML-RPC message which is indented for readability. At transport time the message has all white space characters removed between tags. This particular message is sent to a server to retrieve the temperature for a region, which is specified by the US postal ZIP code [19]. In this example the ZIP string value 10016 is supplied as the argument for the `weather.getWeather` method, which is invoked on the server side.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
    <methodName>weather.getWeather</methodName>
      <params>
        <param>
        <value>10016</value>
        </param>
      </params>
  </methodCall>
```

The XML-RPC response returned by the server contains the method `methodResponse` and a temperature reply of type integer, e.g. `<i4>95</i4>`. The various XML-RPC implementations provide tools to generate such structures as well as to extract parameter information and make the appropriate function calls.

4.6.2 SOAP

The Simple Object Access Protocol (SOAP) [28] is also an XML-based, platform independent protocol to exchange information and request services over the network. Contrary to popular belief SOAP is not officially standardized by the W3C at the time of this writing (August 2002). The first version of SOAP was announced in 1999 and since then four versions have been released. The fifth version (SOAP 1.2) will very likely be included in the W3C XML Protocol Version 1.0 in the near future.

SOAP is an XML based protocol that consists of three parts: an envelope that defines a framework for describing the content of a message and details how to process it, a set of encoding rules for expressing instances of data types, which are defined by an application, and a convention for representing remote procedure calls and responses. Below we show the previous example of a weather service as a SOAP request:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getWeather
       xmlns:ns1="urn:examples:weatherservice"
       SOAP-ENV:encodingStyle=
                "http://www.w3.org/2001/09/soap-encoding/">
       <zipcode xsi:type="xsd:string">10016</zipcode>
    </ns1:getWeather>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

It is obvious that SOAP is slightly more complicated than its XML-RPC counterparts. SOAP uses name space definitions and XML schemes; header elements may be optional or mandatory. The number of SOAP build-in simple types is larger than in XML-RPC and includes types such as `binary`, `byte`, `negativeInteger`, `nonPositiveInteger`, to name a few. In their essence, both SOAP and XML-RPC provide a method name and a list of arguments. For a critical comparison of SOAP and XML-RPC, see Section 4.6.4.

4.6.3 HTTP

Because of its pervasiveness on the Internet, HTTP is by far the most common transport used to exchange web service requests and data. Below we give an example of an HTML message which contains an XML-RPC encoded document in its body. The message was generated by the Grid Migration Service (see 8.1) and intended for a server which is listening on host `vidar2.aei.mpg.de`, port 7010 at the binding `/GPS`:

```
POST /GPS HTTP/1.0
Content-Length: 1745
Content-Type: text/html
User-Agent: GridMigrationServer/0.8
Host: vidar2.aei.mpg.de:7010
Accept: text/html
Connection: Keep-Alive
<?xml version='1.0' ?>
<methodCall>
...
</methodCall>
```

The service handle which is available on `vidar2.aei.mpg.de:7010/GPS` is a central component of the Grid Peer Service implementation, explained in chapter 7. It deserializes XML-RPC messages, tracks requests, passes them on to routines which execute them and ensures that requests are communicated properly.

While HTTP is the most popular transport for SOAP and XML-RPC messages, there are cases when HTTP does not work well with these two encodings. Large size binary data can be base64 encoded and sent by either SOAP or XML-RPC via the HTTP transport protocol. Doing so is time-consuming and hence problematic. The “Blocks Extensible Exchange Protocol” (BEEP)[79] is a recent protocol, which permits simultaneous and independent exchanges of textual and binary data.

4.6.4 XML-RPC vs. SOAP and CORBA

This section briefly justifies our decision for XML-RPC over SOAP and compares the two protocols with CORBA, another widely used framework for distributed applications.

Choosing between SOAP or XML-RPC is important when implementing a service infrastructure. The weather service example above illustrates that the simplicity of XML-RPC is its major asset. It is straightforward to understand, easy to implement and fast to parse. Unlike SOAP its design poses no requirements on the language in which XML-RPC can be implemented. At the time of this writing, approximately 65 different client-server implementations of the XML-RPC specification existed in 33 different languages[4], also reflecting the popularity in the community. SOAP implementations are usually leaning towards object-oriented languages as C++ or Java due to the message complexity. While this is not restricting in a Business-to-Business environment with mainstream hardware, it is a harder requirement in scientific Grid environments with a larger number of “not-so-mainstream” hardware species and supercomputers.

[4] `http://www.xmlrpc.com/directory/1568/implementations`

If the idea of web services is to be moved into a Grid environment, the software which provides the messaging protocol needs to be deployed across a wide range of platforms. It will need to be working on legacy mainframes, supercomputers and proprietary vector computers as well as commodity platforms.

SOAP's greatest feature is its ability to step past XML-RPC's limitations and provide customization at every level of the message. It allows a programmer to specify exactly how he wants to see the message processed and it permits one to define new data types. XML-RPC does not allow self-defined data structures and it is not as type rich as SOAP. There is e.g. no native support to express `NaN` and `INF` values to communicate scientific simulation content.

On the other hand, XML-RPC's robustness and simplicity makes it the ideal choice of data encoding to experiment with web service topologies on heterogenous hardware. All of our requirements were well addressed by XML-RPC, where SOAP would have added a significant overhead. For future implementations of the Grid Peer Service, SOAP and potential competitors should be reevaluated. Since switching between XML-RPC and SOAP is straight forward, giving XML-RPC the preference over SOAP is only a minor obstruction for a future development.

CORBA: The Common Object Request Broker Architecture (CORBA)[24] is an open standard for distributed object computing defined by the Object Management Group (OMG) and has a comparable capabilities like web services. CORBA is an "object bus" and enables a client to invoke methods on remote objects. This is done independently of the language the objects have been written in, and their location. The interaction between client and server is mediated by *Object Request Brokers (ORBs)* on both the client and server sides, communicating typically through an Internet Inter-ORB Protocol (II-OP). The capabilities of CORBA objects are defined using the Interface Definition Language (IDL). The communication protocols used by CORBA for ORB communication include TCP/IP, IPX/SPX, ATM, etc.

With CORBA technology deployed in many industrial disciplines, the web services model is often regarded as a reinvention of the wheel. Gokhale et.al. [45] provide a critical comparison of both technologies. A key difference between CORBA and the Web service technologies (UDDI, WSDL, SOAP) is that CORBA provides an object-oriented component architecture unlike the message-based web services (and despite of SOAP's name). CORBA has a rather tight coupling between the client and the server, where both must share the same interface and must run an ORB. In the web service model everything is decoupled, a client sends a message and receives a reply. Web services allow for thin-clients, while CORBA requires a large footprint on all participating sites. While CORBA implements its own reliability and scalability policies (e.g. "Load-balancing CORBA"), web services are not burdened with this responsibility. Instead, it is left to the application servers, which often bring their own, native fail-over strategies. On the social side, web service evolution is experiencing tremendous community contributions, while CORBA has consolidated into a rather inert, industrial product.

An important observation concerning CORBA and web services is that whatever can be accomplished by CORBA can be accomplished using Web service technologies and vice versa. In particular, one can implement CORBA on top of SOAP, or SOAP on top of CORBA. SOAP and CORBA are not exclusive but rather should be seen as complementary technologies that need to coexist.

4.7 Discussion

We chose not to go with any of the existing UDDI and WSDL models since they are all focusing entirely on the service aspect. We are developping and experimenting with a comprehensive information model that describes more than an isolated component of a Grid entity. Instead of regarding a service as self-contained information and storing it as a WSDL document, we take a different approach and introduce Grid Objects (Chapter 5) as a handle to entities on a Grid. In this model, a service is only a special aspect and is seen in conjunction with e.g. file properties, resource characteristics on a machine. We use Grid Objects to communicate content with migration and other services. Section 5.11 provides a comparison of the Grid Object approach to the service-centric strategies like WSDL or OGSA. Instead of using the service-focussed UDDI, we extend the idea of a *service registry* towards an *information registry* and introduce the Application Information Service (Section 7.1). This service acts as a “data warehouse” and stores Grid Objects, which contain arbitrary information on services, files, resources, etc. Information can be retrieved and deposited by applications, services and users.

While the idea of web services has been around for a while, it has just recently been put by into the context of Grid computing – most prominently by OGSA. In the “old days” custom applications and protocols existed side-by-side and Globus was the most embracing solution. The fusion of P2P and web services combines fault tolerance and data redundancy with service abstraction. We believe that this combination is a promising way to interconnect Grid middleware and to achieve the necessary failure tolerance to operate in multi-organizational, global Grids.

Chapter 5

Grid Object Description Language

This chapter defines an information model for specifying a general handle to entities on a Grid. These entities or objects represent the components of the Grid infrastructure or applications that make use of it. This notion is called *Grid Object Description Language*. The data model is motivated by the experiences and experiments conducted with real applications in heterogeneous Grid environments.

An "application-centric" network of services requires a precise definition of objects to be able to communicate content. While sophisticated data models exist to describe and characterize individual aspects (e.g. resource or services), we show that a comprehensive description, which combines *multiple* aspects, is needed to give an adequate representation of objects on a Grid.

We start this chapter with a thorough motivation of the ***G**rid **O**bject **Des**cription **L**anguage (GODsL)* as a hybrid information model to objects on a Grid. We then show how the generic and extensible GODsL unifies the various aspects of Grid entities into a single data structure. GODsL is a conceptual model and is not bound to a specific implementation. Appendix A introduces a C-implementation of a GODsL toolkit which is used in the implementation of the migration and spawn environment.

5.1 Motivation for Grid Objects

We illustrate with several examples, what we mean by "object on a Grid" or "Grid entities":

Grid Objects Motivation, Example 1: The information on the name and directory of a data file is only sufficient if we know on which physical machine the file resides. However, we have no information on how we can access that particular computer, we are not able to look at the file, copy it or treat it in any other way.

To let services work with objects like files, compute capacities or migratable applications, a common data model is needed, which reflects the different aspects of these objects. Such a model needs to provide a structure that combines all of its properties. Example 1 demonstrates that for an application-centric grid environment, information of this kind needs to be available within a single data structure and not spread out over several different and unrelated data constructs.

Grid Objects Motivation, Example 2: A file located on a computer which is connected to the internet can e.g. be accessed through copy operations. The standard grammar of addressing such a file, which is owned by a user monroe, is: `monroe@astro.umsl.edu:/home/monroe/MyArticle.ps`. The same file can be made accessible through a web server, in which case the grammar (now tied to the HTTP protocol) becomes: `http://www.umsl.edu/~monroe/MyArticle.ps`.

The above example illustrates two user-centric expressions: the first expression does not make a statement on how to access the file, but it conforms to the syntax of `rcp`, `scp` or `GSI-scp` copy operations. A UNIX familiar user will recognize this immediately and use the appropriate method. The second expression gives the URL of the file and indicates that the transfer is made through HTTP.

Both expressions hide details like port information and authentication. An *ssh* connection is usually made through port 22 while incoming HTTP communication is channeled through port 80; copy procedures between machines usually require password or pass-phrase authentication. Our information model must be capable of associating the file `MyArticle.ps` with one or more access methods (e.g. ssh, http) and it must accommodate additional information such as port ranges.

Grid Objects Motivation, Example 3:

1. The file mentioned in Example 1 may have been generated by an application. Unsurprisingly, the machine which hosts the application is identical with the machine location of the file.

2. This application has a minimal *resource requirement*, e.g. number of processing elements. Another computer may provide one or more *resource capacities* (e.g. the main capacity partitioned by batch queues). The host can execute the application if and only if the hardware's resource capacity is greater or equal than the application's resource requirement.

3. If such a resource match succeeds, the applications can be hosted on that particular hardware or migrated to it. The description of the application is identical, but the host information changes.

In Example 3 we can identify four core fundamental components of an object, which are its service, hardware, file and resource properties. It is obvious that the components which assemble an object come with a high degree of reusability. For example, components can be inherited, as seen in Example 3.1, where the file's machine description is copied from the application object. Components of the same type can be put in relation, as shown in Example 3.2, where two resources characterizations are compared. Components can be added or replaced, as featured in Example 3.3, where the description of the hardware is added or replaced as the application starts or migrates, respectively. It is clear that we need to supply a data model which integrates very different aspects and allows one to swap, replace and copy its sub-structures. It must be scalable to allow the proper description of complex objects.

Legacy Services in Grids: Condor, PBS and LSF, to name a few, provide services to manage computer resources and to submit jobs to particular batch queues on a machine. Compared to the web services we refer to them as "legacy services" — which is not meant derogatorily: While web service protocols like SOAP have capabilities to translate the request vocabulary, legacy applications come with a proprietary syntax for describing resources and are illiterate regarding other systems. Each application has special abilities when defining a resource but to the greater extent all describe common features like memory, number of processors, etc. The description language we strive for must allow a mapping between these common features and act as an intermediary when dealing with such services.

We have realized that the legacy services which are deployed throughout Grid environments simply do not come as self-descriptive web services that conform to WSDL or UDDI. To integrate them we must translate on a syntactical level first – before we can attempt to interface their functionality. Our descriptive language for Grid objects has to be elementary enough that its grammar can be mapped to an already existing vocabulary of a resource management system. With our approach we have no ambition to describe every single capability. However, by expressing the most common features, we are able to join together a significant number of legacy software systems.

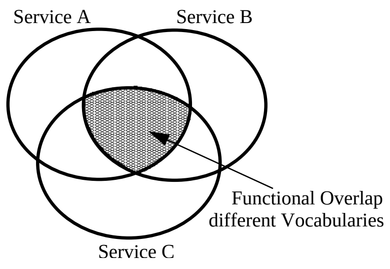


Figure 5.1: Different (legacy) service applications for a common purpose (such as batch submission) show a functional overlap. However, they do not share a common syntax. Grid Objects provide a mapping between different vocabularies and help to access the same functionality in different tools.

5.2 Definition of Grid Objects

We propose *Grid Objects* as a general description of objects on a Grid. A Grid Object is a collection of sub-objects, termed a *Container*, which hold one or more *Profiles* that reflect the different properties of this object. The container structure in a Grid Object is optional and depends on the background situation that is described. We call the structure to describe such an entity the *Grid Object Description Language*.

For our purposes we define a Grid Object as follows:

$$GridObject \rightarrow \textit{UID}\ \ \textit{Label}\ \ \{MachineCont.\}\{ResourceCont.\}\{ServiceCont.\}\{FileCont.\}$$

A container is a wrapping structure which holds one or more profiles. A profile in a container may be optional.

$$Container \rightarrow \textit{UID}\ \ \textit{Label}\ \ [\textit{Profile}\#1]\ldots[\textit{Profile}\#n]$$

The profile is the fundamental building block of a Grid Object. It represents the object's properties through a fixed set of attribute/value pairs, which is specific to each profile type. The presence of the specific set of attributes is mandatory, the associated values are not.

$$Profile \rightarrow \textit{UID}\ \ \textit{Label}\ \ \{\{\textit{Attribute}\}[Value]\}$$

The attributes for a unique identifier (UID) and a label are available at all structure levels (object, container, profile); associated values are optional.

Classifying Profile Types: The classification into the fundamental types of machine, resource, service and file profiles is inspired by our underlying migration scenario, whose communication content we intend to express with this data model. For other scenarios, different profiles may become important while others can be omitted. We believe that the modular Grid Object hierarchy allows accommodation of most cases. In section 5.10 we suggest several important additions to the Grid Object information model.

5.3 Service Profile

A *service profile* is a description of a functional ability which is available on a machine or provided by an application, usually through a port or range of ports. We aim at describing modern web services as well as traditional interactive commands like secure-shell based access to a hardware or PBS based queue submission system. The key attributes, which are used to describe such a profile are

Type	A classification for the service. The service class states a common functionality, which may be implemented in different ways (e.g. a file copy method).
Label	A human readable label, describing this particular service.
Operation	If the service is not a web service, operation states what command needs to be executed on a machine.
Binding	Used to flag an RPC typed web service. It holds the Binding of a URL under which a web service is reachable. The URLs hostname can be derived from a machine profile.
Method	For an RPC typed service, the name of the remote procedure call which invokes this service.
Transport	A definition of the transport protocol. For web services, most transport methods are HTTP based. Other transport mechanisms are e.g. secure shell, HDF5 data streams or LDAP based database queries.
UID	A unique ID to identify this particular service. This ID is used to distinguish services within an application. The ID should be unique within the scope of the service environment.
Port[]	An array of integers holding port numbers, e.g. port 22 for secure shell based access to a machine.
PortLabel[]	An array of labels to distinguish the different ports.

One or more service profiles are collected into a *Service Container*. The service container rolls several service definitions into a single structure. Such a container may e.g. list all access methods (e.g. ssh, rsh, HTTP, Globus gatekeeper) of a computer.

ServiceProfile[ ]	An array of service profiles
Label	A human readable label describing this collection of services.
UID	An unique ID, which identifies this collection of services.
Count	This variable which holds the number of attached profiles is used to ensure the consistency of the number of attached structures.

Service Profiles and WSDL: At the risk of being repetitive: there is a fundamental difference between a Web Service and a Grid Object: a (web) service is an isolated aspect of a Grid entity and Grid Objects correlate the different aspects into a single representation. Grid Objects and WSDL were designed with different intentions: WSDL is a language to describe web service functionality, like the operations that a web service can support, its arguments and return values. Grid Objects are a unifying handle to entities on the Grid. One aspect of such an entity are its service capabilities. The service profile was designed to accommodate non-web services, like user applications and legacy infrastructure as well. The descriptive capabilities of WSDL are superior to service profiles. However, it is not our intention to reinvent the “service-description wheel”, but rather to experiment with a more embracing data model.

Possible ways to fully incorporate a web service description in a service profile include the duplication of WSDL functionality or storing a complete WSDL document. Both options are clearly problematic. Instead, using the service profile as a handle to a WSDL document appears to be a far more practical approach if need comes up. Such a service profile can then be used to retrieve the proper WSDL document.

Classifying Services: The service profiles provide room to classify a service but do not prescribe a set of pre-defined classes, which has to be obeyed by a service. Service profiles do provide a system to sub-categorize a class. We found that a classification system needs to be consistent within the pool of applications which provide and consume the service. Obviously, a service declared to be of type "copy" may not delete the file. There is currently no way to "guarantee" that a service will do what it pretends to do. The service verification becomes even more challenging if the differences are more fine grained. For instance, "copy" and "move" operations yield the same result (a file on a new location), but have different side effects (removing vs. keeping the original).

Migration Service Classes: Within the migration service environment, we settled on a set of service classes, which we found useful and which expressed all required services. Our classes are:

`copy,delete,move`	These service classes classify copy, delete and move operation for files on the Grid. The GPS implementation uses `ssh`, `rsh`, `GSI-ssh` and `sftp` to perform these file operations.
`shell`	This type summarizes all services, which allow shell access on a remote machine. Remote shell access is carried out e.g. through `ssh`, `rsh` and `GSI-ssh`.
`submission`	The submission service class was introduced to provide a category for submission-only access, e.g. through Globus, PBS, LSF.
`migration,spawn`	These two classes provided a category for the migration and spawn services.

Other situations will most likely require a different classification system. This is easily possible, since GODsL does not promote certain service classes.

5.4 File Profiles

File profiles are used to describe the properties of files and directories. In a user-centric environment, the file's name and directory often suffices to manage them on a single machine. In an automated web service environment, additional information is useful e.g. to avoid name conflicts, to let a service choose the best method of file transfer, or to describe a single logical data file made up by separate chunks. We motivate a file profile with a typical example of simulation I/O:

File Objects, Example: A parallel simulation performs periodic checkpoints to save the application's state as a protection against failure. To speed up checkpointing, the application writes one data chunk per processor. The full checkpoint is composed by all files. After a migration to a single processor machine a new checkpoint is drawn and the checkpoint is now represented by a single file.

To describe such situations we introduce a data structure called *File Profile* which describes a single file. A file profile does not posses information on the machine that the file is located on, nor does this profile contain information on a higher-level data composition, like the logical checkpoint in the example above, which is made up from different physical files.

Name	The name of the file as given by the user or the generating application and as it is found in a directory on file system.
Directory	The directory in which a file can be accessed on a file system. Name and directory provide a unique addressing scheme on a computer. They cannot be used to identify a file in Grid environment, since different directory/file pairs may be identical on distinct machines.
UID	An ID, which is used to provide a unique description. It can be used to distinguish the file in a Grid environment – even if names of the same name exist. Provided that the uniqueness is preserved, an ID can be used to express relationship information for a collection of files, e.g. the number of checkpoint chunks in the example above.
Type	A classification which permits a categorization of file. The classification is open and not defined within the GODsL.
Label	A human readable label, given by the user or the generating application. This label is not intended to be interpreted by an application, it has only illustrative purposes.
Size	The size of the file. This information can e.g. be used to determine if multiple small files should be combined into a large archive before transmission to avoid overhead.
SizeUnit	Unit of the size attribute.
Date	Date information for this file.
Compactification	A floating point number indicating the compactibility of the file, which ranges from 0 for not compressible to 1 for highly compressible. This factor would ideally be provided by the generating application since it knows best about its origin and internal structure. A second approach derives this factor from related files (e.g. the compression properties of a checkpoint are identical to others of the same origin).

To collect multiple files to a logical unit, the file profiles are collected to a file container.

FileProfile[ ]	An array of file profiles
Label	A human readable label describing this collection of files.
UID	A unique ID, which identifies this collection of files.
Count	This variable holds the number of attached profiles and is helping to ensure the consistency of the attached structures.

File Identification: Files are intuitively described by their name and directory. This identification is risky because it relies on the user to supply a unique name – a modus which usually works in a small setting, like on a local PC. For a Grid environment with many participating sites and automated services the limitations of this scheme are obvious, e.g. if files are automatically transferred to storage facilities. In addition to the name/directory which is unique per file system, we the tag the file profile with an unique identifier. This UID can also be used to express relationship properties among related files, e.g. in parallel-I/O scenarios. The UID scheme helps in distinguishing files as long as all services honor this scheme. The approach becomes problematic if a service relies on legacy copy operations. While the service honors the uniqueness, it cannot avoid that the underlying copy operation overwrites a file with another one, because both have different UIDs but share the same directory and file name.

It is difficult to waterproof the UID file concept for legacy environments. A workable, but questionable approach is to append the unique ID to the name of the file. This works in an automated environment, but complicates the manual handling of files. The data grid community [27] is using a *replica manager*[1] which performs a mapping among the physical and logical file name that is globally unique.

File Properties: Important additional information for large files are their sizes and compactification properties. The latter can e.g. be used to indicate an ASCII formatted file, which compresses well against a binary file, which is already compressed. Both pieces of information allow a service to pick the best of all available transfer methods or to rule out certain machine targets, whose bandwidth is too small to achieve a file transfer in a specified time.

5.5 Machine Profiles

A *Machine profile* contains the information which is obligatory to address the hardware on the internet, e.g. through an IP number. The machine profile introduces the location aspect of an object as it is used in conjunction with other profiles. The machine profile reflects the properties of a single device in the internet. It does not contain information on its resource characteristics, like memory, type of machine, etc. The machine profile provides specific information on:

Hostname	A fully qualified hostname, if available.
Domain	The Domain name of the machine, if available. Quite frequently machines are configured in a way that the application cannot determine the machine's full domain name.
IP	IP number, if hostname is not available. This is often the case on cluster systems, where single nodes do not come with hostnames.
Type	Allowing for a categorization of hardware. No classification is enforced.
Label	Human readable information string describing the machine.
UID	unique ID. It does not suffice to use the IP number as the sole identifier. This approach would allow duplicates if machines are hosted on private networks.
Username	Username under which the machine can be accessed.

The *Machine Container* is the wrapping structure for one or more machine profiles. It is used to group many hardware devices into a logical unit. The attributes of a machine container are:

MachineProfile[]	An array of machine profiles.
Label	A human-readable label describing this collection of machine profiles.

[1]Data Replication Research Group (REP): `http://www.zib.de/ggf/data/rep.html`

UID	A unique ID identifying this collection of devices exclusively.
Count	A number of profiles in this machine container. The count variable is used to ensure consistency when profiles are attached to the container.

Machine container can be used to describe a network of workstations or the participating machines of a meta-computer, where each participant is expressed through a profile. The machine profile and container structure does not make a statement on the granularity at which a machine description is carried out: for a cluster computer, a single machine profile for the head node can be used to describe the full machine. A comprehensive approach uses a profile for each cluster node and groups all nodes in a container. Which style is chosen, depends on the situation: If the cluster requires that individual nodes are listed in a machine file for a mpirun statement, the detailed approach stores the necessary information.

5.6 Resource Profile

The *Resource Profile* holds information, which characterizes the compute *capacities* or *requirements* of a Grid Object. Important attributes of a resource profile describe the number of processing elements, memory or information on the operating system.

The reason to treat the resource characteristics separately from a machine profile is justified by the fact that resource profiles are not necessarily tied to a hardware device. Resource profiles can be used to characterize applications or abstract resource situations as well: An application may provide a resource profile to interested parties to relay information on its *resource consumption*. In this context the resource profile is not attributed to a hardware (machine profile) but to an application, which is described through its location (machine profile) and functionality (service profile). Multiple resource profiles may be used to characterize the different batch queues which partition the total compute resource of a machine, specified through the machine profile of the head node.

The information in a resource profile is usually supplied through a resource monitoring system, which watches the resource state of an application or a machine. Performance-API [65] is an example of a software package, which returns information on memory usage or floating point performance of an application. The Metacomputing Directory Service (MDS)[25] is an example of a service suite, which observes and reports the resource situation on computer hardware. All of these systems come with a proprietary syntax on how to specify the resource requirements or constraints, as shown in Figure 5.10, page 54. While our resource structure is not able to describe the full functionality of every single resource system, we define a vocabulary, which is large enough to provide a mapping to the syntax of most resource systems. In Figure 5.1 we sketch this syntactical overlap.

The attributes in resource profile are able to represent the resources for a Grid migration service. They may be incomplete for other situations or require adaption. An expansion by adding attributes or new profile types is straightforward.

Processors	The number of processing elements.
Memory	The amount of memory described in this profile, e.g. the memory requirement of an application or the resource capacity of a machine.
Memory Unit	Memory units (G/M/K = Giga/Mega/Kilo).
Operating System	Description of the operating system, as reported by the `uname` command.
Machine Type	A set of keywords to distinguish a *workstation*, *cluster*, *vectormachines* or *supercomputer*.

CPU speed	Frequency of the central processing unit.
CPU Speed Unit	Unit of the CPU speed in Hz (G/M/K = Giga/Mega/Kilo).
CPU Load1/5/15	CPU load during the last 1/5/15 minutes. These attributes are directly derived from the MDS database entry and used to monitor the interactive execution of programs. They are useful for experimenting with the interactive execution of programs. This data is not relevant if the execution is scheduled through a batch system.
RunCmd	A string holding the proper MPI run-command for parallel environments (e.g. mpirun -np , mpprun -n, poe, etc.).
UID	A unique ID to describe this resource. Like all UIDs, it can be used to express relationship information between related resources.
Label	A human-readable label describing a resource profile.

Multiple resource profiles are grouped within a *Resource Container*. The attributes in such a container are:

Resource Profile[]	An array containing one or more resource profiles.
UID	An unique identifier which is used to discriminate between different resource profiles.
Label	Human readable label holding information on this resource configuration.
Count	Number of profiles in this resource container.

5.7 Grid Objects

The different profiles that we introduced above can now be combined to provide a unified description of the different aspects of objects on the Grid. In this section we describe Grid Objects as the main objects of the *Grid Object Description Language* and construct specialized objects with regard to file and resource descriptions.

The Grid Object structure collects the various containers to a single entity. Container information is optional, they are attached depending on the situation which is expressed through the Grid Object. The Grid Object consists of:

Machine Cont.	Linking in an optional machine container.
Service Cont.	Linking in an optional service container.
File Cont.	Linking in an optional file container.
Resource Cont.	Linking in an optional resource container.
UID	An unique ID which is used to identify the Grid Object.
Label	Human readable label holding information on this Grid Object.

Multiple Grid Objects can be collected to an array of Grid Objects, called a *Grid Object Container*. The complete structure of a Grid Object is shown in Figure 5.2.

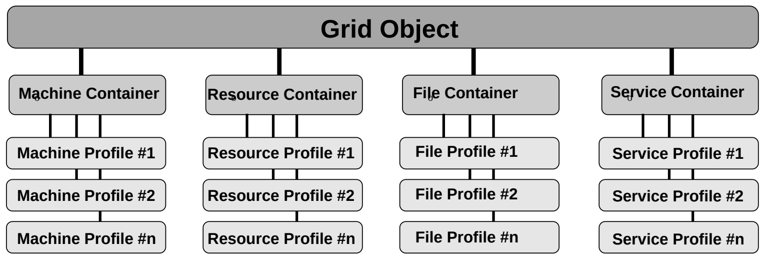


Figure 5.2: The complete structure of the Grid Object data model, consisting of machine, resource, file and service profiles as the fundamental components.

Unique Identification Numbers (UID): The sub structures of a Grid Object can be distinguished through a unique identifier. How a UID is provided is not defined by GODsL, but left to the application. Generating globally unique IDs is an everlasting problem in Computer Science. If GODsL objects are used in an enclosed application environment and all participants agree on the same scheme, pseudo unique ID will work well. Such IDs can be e.g. be generated by combining the process ID and the current time. This scheme is used to set the UID for the advanced migration scenarios.

For truly globally unique IDs, the deployment of a UID server is a better solution. A client contacts the UID server and receives an ID. Such a process must operate reliable. A redundant server strategy as described in Chapter 4 can be used to accomplish this reliability. Distributed UID server require inter-communication to define the UID ranges which they operate on. We have no room to elaborate on this problem, but would like to mention the service monitor system, described in Section 9.4.3 as a framework to supervise applications or distributed service instances and restart them if necessary.

5.8 Special Grid Objects

This section introduces a number of specialized Grid Objects, which provide a comprehensive description of files, resources and networks interconnects on a Grid.

5.8.1 Grid File Object

A *Grid File Object (GFO)* defines a Grid Object, which describes the location of a file on a Grid. A file object requires a machine and file container, the existence of a service information on how to access the file is optional. Later, if the files are to be accessed, this than one network profile information is indispensable, and it must be attached in time to access the file.

$$GridFileObject \rightarrow \textit{UID}\ \ \textit{Label}\ \ \{MachineCont.\}\{FileCont.\}[ServiceCont.]$$

5.8.2 Grid Resource Object

A *Grid Resource Object (GRO)* defines a Grid Object, which describes the the resource aspects of a machine or application. A resource object mandates at least one resource profile in a container and either a service container to characterize the application or a machine container to describe the hardware. Multiple resource profiles per container are used to describe batch queues which partition the total compute capacity of a machine.

$$\begin{aligned} GridResourceObject \rightarrow & \textit{UID}\ \ \textit{Label}\ \ \{ResourceCont.\} \\ & \big(\{MachineCont.\}\ \wedge\ \{ServiceCont.\}\big) \end{aligned}$$

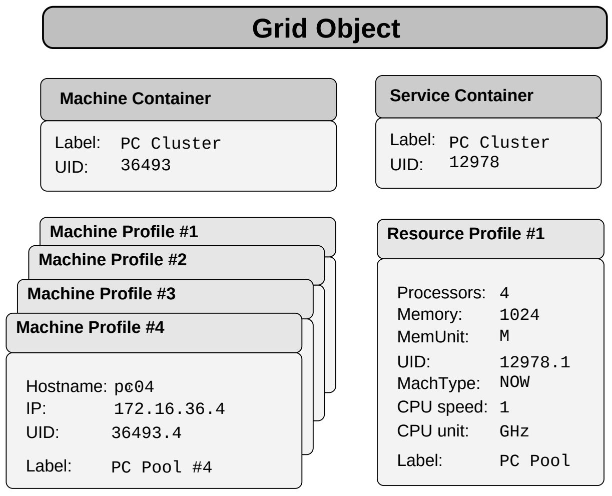


Figure 5.3: A simple network of workstations (NOW) is made up from 4 PCs. The machine profile describes the hardware of this cluster. A single resource profile is used to characterize the combined computing capabilities of the cluster.

5.8.3 Grid Network Object

A *Grid Network Object* defines the network quality between two network endpoints. The edge between the endpoints is provided through a network container which holds a network profile. The actual endpoints are defined through two machine profiles, which are located in a single container. The machine profile's *type* attribute allows do introduce directed bandwidth statements, distinguishing e.g. between upload/download characteristics. While the machine container must hold two profiles only, the network container may hold more than one network profile.

$$GridNetworkObject \rightarrow \mathit{UID}\ \ \mathit{Label}\ \ \{NetworkCont.\}\{MachineCont.\}$$

The Grid Network Object is not part of this thesis, but well be followed up on in a future project, see Section 5.10.

5.9 Grid Object Examples

In this section, we pick typical situations found in Grid computing or migration environments and use the Grid Object Description Language to provide a proper representation of the information involved.

5.9.1 Machine Constellations

We give a number of examples that illustrate how machine profiles can assembled to provide an adequate picture if individual machines are part of a more abstract machine setting, like in cluster or meta-computer.

Workstation Cluster: In Figure 5.3 we illustrate the Grid Object for a loosely coupled network of workstations (NOW), made up from simple PCs. Each machine is described by a machine profile, which identifies the hardware by its IP number. The clustered structure is preserved in the machine container. The granularity at which a cluster can be described is discussed in Section 5.5.

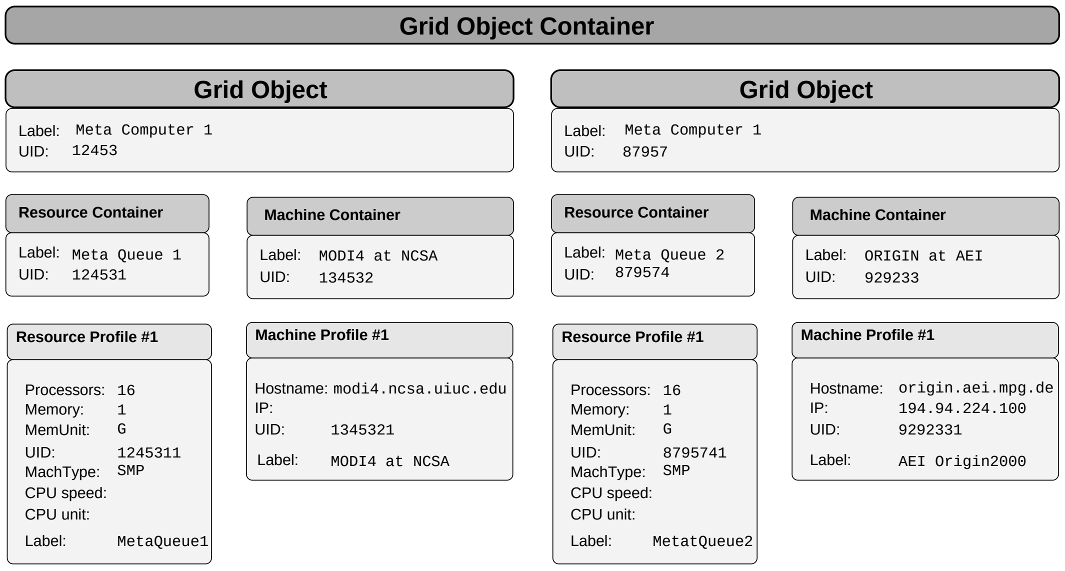


Figure 5.4: A meta-computer is a abstract computing resource built from several physical machines. The example illustrates a meta-computer which is assembled by two O2K at the Albert-Einstein-Institute (AEI) and the National Center for Supercomputing Applications (NCSA).

Meta-Computer: Figure 5.4 shows the Grid Object representation of a meta-computer. In a distributed meta-computing environment, multiple geographically distributed machines are combined to a single, virtual Grid computer. In this example, a meta-computer is assembled from two machines at the Albert-Einstein-Institute (AEI) and the National Center for Supercomputing Applications (NCSA).

5.9.2 Grid Service Objects

Service profiles specify how an object can copied, accessed or manipulated in any other way. We give examples where service profiles are used. The service profile itself makes no statement on the location of the service. The location (= hardware) of the service is defined by attaching a hardware profile.

Simulation Services: The first example describes a proprietary service provided by a simulation based on the Cactus framework. The application has opened ports to let scientist introspect the current simulation state [57]. The application has generated its own web page, which is accessible through port 8010. It streams simulation data to external visualization software. Here, port 8015 is used to stream the data while port 8016 steers the geometric shape of the extracted data. It is not necessary that every web service understands the syntax behind this port configuration, an program which understands service type `IOStreamedHDF5` would know how to deal with the port setup described by Service Profile #2.

Machine Access Services: In the following example we characterize the GSI-ssh service. GSI-ssh is the ssh version adapted to the Globus Security Interface [38]. The GSI-ssh daemon usually operates on port 2222. `gsissh` and `gsiscp` are two application to gain shell access and copy files, respectively.

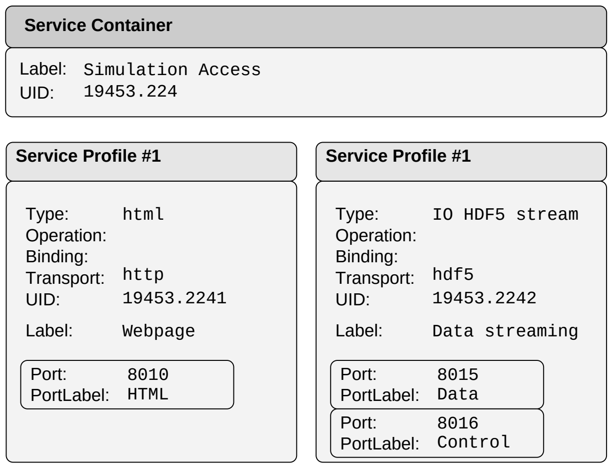


Figure 5.5: A simulation allows scientist to introspect the ongoing simulation. This code generates a web page by streaming HTML code through port 8010 (Service Profile #1) and sending HDF5 formatted data to an external visualization client like OpenDX (Service Profile #2). The HDF5 data is streamed through port 8015, while control parameters like down sampling are accepted on through port 8016.

Migration Services: The example in Figure 5.7 contains the description of a migration service. The service can be contacted on port 7010 via HTTP transport, the name of the associated RPC is `migrate`. A machine profile (not shown) contributes the hostname.

5.9.3 Grid File Objects: Single File on a Single Machine

Figure 5.8 illustrates a Grid File Object, which holds the description of a single file on machine. The file properties, like size and compressibility are captured by the *file profile*. The *machine profile* contributes the location of the hosting machine and the *service profile* lists the supported methods to access the file, here through a GSI-scp command. Additional *service profiles* can be listed if the machine's environment offers these access types. When a file is moved to a new host, machine and service profiles are replaced; if the file is copied, a duplicate of the Grid File Object is created and updated according to the new host.

5.9.4 Grid File Objects: Multiple Files on a Single Machine

Large data files in parallel applications are often brought to disk through parallel-IO methods to accelerate the process of data writing. Parallel-IO ideally requires constant IO time as the number of processors increases, but it generates multiple files, as sketched in Figure 5.9. To express this situation, we can accommodate multiple file profiles in a file container, thereby preserving the information that they are all part of a logical unit. In Figure 5.9 we show the expanded, single-file example, in which the hosting machine and access services stayed the same.

5.9.5 Grid File Objects: Multiple Files across Multiple Machines

An often mentioned scenario in Grid computing is the execution of a single application across multiple supercomputers. The data, which is generated through parallel-IO techniques now resides on multiple machines. For example, a logical file is composed of 10 data sets, where six data sets are generated on machine *A* and the remaining four are residing on machine *B*. Both machines may have *different*

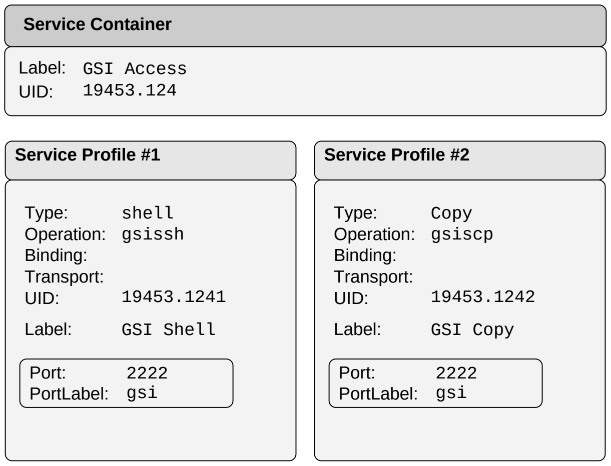


Figure 5.6: A description of GSI-ssh typed access to a computer. The Globus Security Infrastructure provides a ssh based access to shell and copy functions, usually through port 2222.

access methods. We can provide a proper description by creating a Grid File Object for A and B. The two Grid objects together yield to full and accurate description of the distributed files. If the files are physically moved to a single w machine, a new file description is obtained by creating a Grid Object with the machine and service profile of the new computer and joining all file profiles.

5.9.6 Grid Resource Objects: Application Requirements

The resource profile can be used by an applications to characterize its memory and CPU requirements. This information can be gathered automatically at runtime or manually. The resource consumption characteristics of an application is used to determine an appropriate compute capacity before a application is relocated. If a matching machine is found, the Grid Object description of the application can inherit the machine profile. The resource matching process is described in the next section.

5.9.7 Grid Resource Objects: Resource Identification and Evaluation

Before any job is executed on a supercomputer, a three stage process needs to be completed, which is traditionally done by the user, interactively and intuitively:

1. **Resource Identification:** The user determines the resource requirements of his application, typically through educated guesses or trial and error. Secondly, the user has to familiarize himself with the compute capacities on the machines in question. This knowledge is usually gathered by reading up on the site's batch submission configuration.

2. **Resource Evaluation:** The user decides for a particular supercomputer, where he choses a queuing system whose constraints will not be violated during the runtime of the program.

3. **Resource Request:** The user requests the resources, usually by filling out a batch submission script, in which he states his requirements, like number of processors and memory. The user has to obey the particular queue syntax. He finally submits the job.

Since we can express the diverse resource requests and constraints as profiles in a Grid Object, we can compare them to determine suitable resources for migrating applications. In Figure 5.10, we illustrate

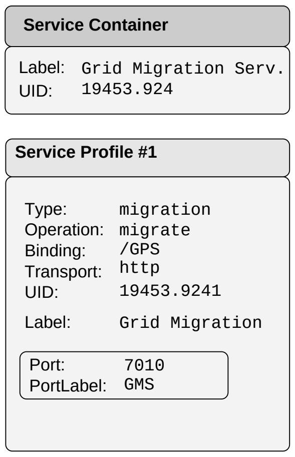


Figure 5.7: A migration service describes its service through a service profile. The migration service is part of the Grid Peer Services described in Chapter 7. The complete URL (e.g. `origin.aei.mpg.de:7010/GPS`) can be derived from the port number, binding and the hostname in the machine profile (not shown).

this automated, three stage process, where the resource identification is shown to the left, followed by the evaluation stage which compares and selects a resource. Finally, the appropriate batch submission scripts are filled out.

Condor Class-Ads: In a service environment, the resource matching and selection process is carried out by an application instead by a user. One of the advanced decision making system is *Classified Advertisements ("Class-Ads")* [22], used by Condor. Applications and resources advertise their characteristics in a special data format and a matchmaking process selects those pairs which fulfill the specified relation. Section 3.1.2 describes Class-Ads in detail. In Figure 5.10 we outline the process of matching the requirements of an applications with the resource constraints provided through the queues on a supercomputer. Figure 5.10 sketches the translation of different vocabularies, which are used by the various systems.

- **Resource Identification** The first stage of a submission process concerns the identification of resource constraints on a machine and resource requirements of the application.

  1. *Resource constraints*: In Figure 5.10, on the left side, the upper resource container holds profiles, which describe the resource characteristics of the queues that partition the total compute capacity of a machine. Such queue information can be obtained through a query of resource databases like the Meta Directory Service (MDS). The MDS is discussed in section 3.1.1. For each host, which is reported as a potential execution host, we generate one Grid Resource Object, which in turn may hold more than one resource profile. These profiles state the resource *constraints*. The first gray box in Figure 5.10 illustrates the mapping of the MDS vocabulary to the profile structures.

  2. *Resource requirements*: the lower profile in Figure 5.10 is reflecting the application resource needs: the program demands 500 MByte and 64 processors. The data can be supplied manually by the user, who may for example specify the maximum amount of experienced memory usage or restrict the architectures to those that he has precompiled executables for. In advanced application information on the memory requirements is ob-

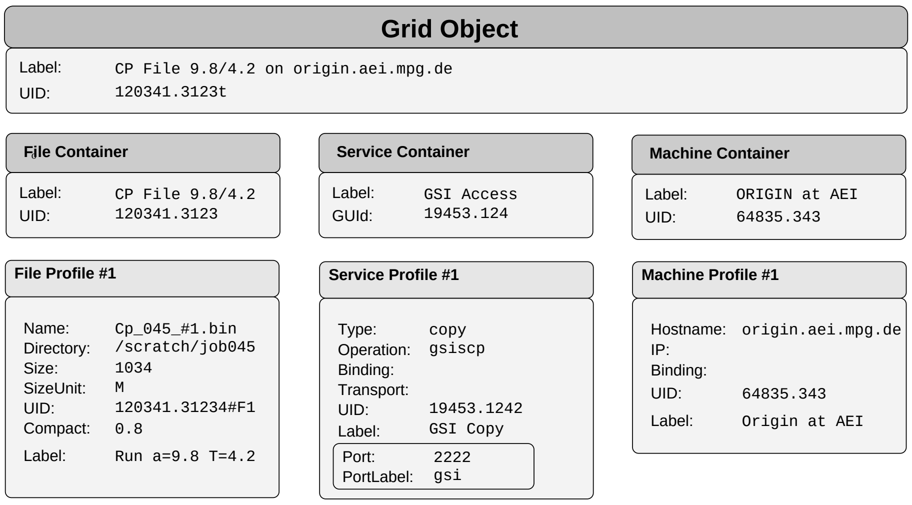


Figure 5.8: The access of a single file on a single compute resource can be described by file, service and machine profile. Together these three profiles describe the file's properties, the access methods supported on the hardware, as well the hardware itself.

tained by instrumenting the executable and extracting resource consumption rates at run-time. The data is expressed in a resource profile with designates the *requirements*.

- **Resource Evaluation:** To determine which resource provides the best compute capacity for a given resource constraint, we prefer using existing technology, like Condor Class-Ads: A migration service rewrites the resource profile of the application and the resource profiles of the available resources as Class-Ads. The Class-Ad parsing algorithm compares the job requirement to the constraints and returns a match if possible.

- **Resource Request:** When a match is found, the service fills out the batch submission script, using the appropriate syntax. The contents of application's resource profile is now mapped onto the vocabulary of the batch submission systems. We show this translation for PBS and LSF in the bottom part of Figure 5.10.

This simple example already involved the four resource "dialects" and data formats of MDS, Class-Ads, PBS and LSF. For example: the amount of memory on a system is expressed as `physical-memorysize` (MDS), `Memory` (Class-Ads, user defined), `#BSUB -M` (LSF directive), `#PBS -l mem` (PBS directive).

We have shown in this section that the Object Description Language permits us to converse with already existing software packages and grid middleware and can adapted to integrate upcoming Grid technology as well. Grid Objects fulfill the important requirement of compatibility and enable third-party software to interoperate.

5.9.8 Dynamic Object Composition

The different profiles which are composed into a Grid Object do not need to come from a single source or at the same time. It is even unlikely that a single application is knowledgeable about all aspects of a Grid Object. Different applications may contribute different information at different times. As an example we illustrate a situation, where two different instances contribute to a common Grid Object, as illustrated in Fig 5.11: A simulation creates a Grid File Object, which identifies the location of a file.

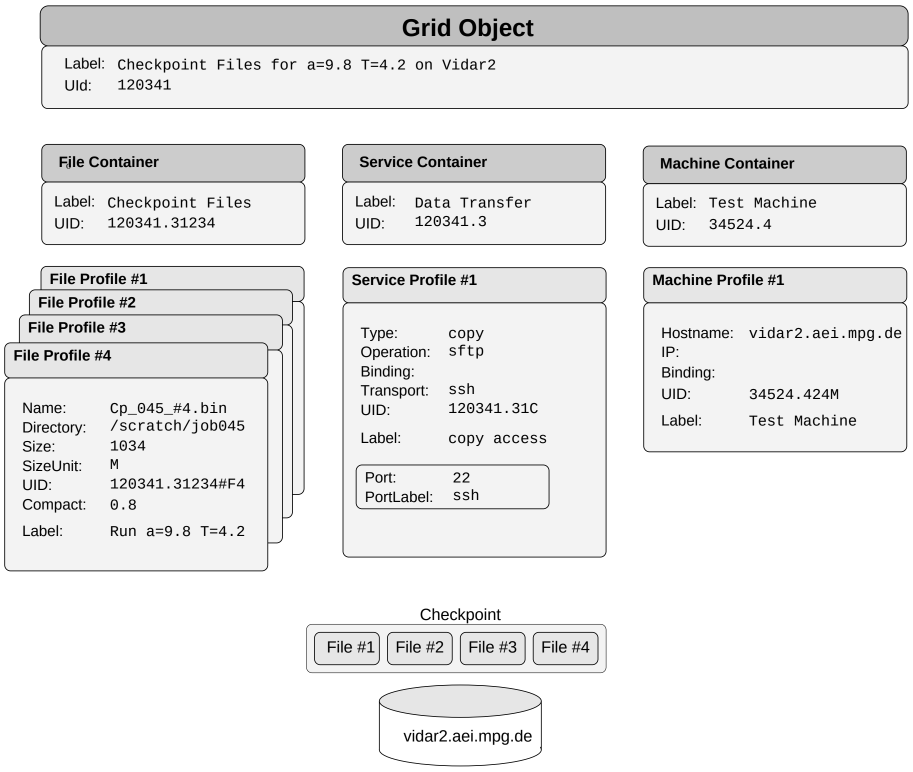


Figure 5.9: Multiple files on a single compute resource. Multiple file profiles describe the different file chunks. The service profile determines the access method, while the machine profile has the information on the hardware.

A service profile for the access methods is not provided, because the application has no knowledge. If the file is about to be accessed e.g. through a file server, the machine profile is used to query an information database like the AIS (Chapter 7) on the supported access methods for that host. The completed Grid File Object holds now a full prescription on how to access the file.

5.10 Future Research

In this section we give an overview of extensions to the Grid Object Description Language and potential synergies with other research projects. The data model has enough structural capacity to hold extra components without becoming to heavy to use. In the following paragraphs we list two extensions to express additional Grid properties.

Network Profiles: The current Grid Objects do not support the characterization of network performance. However, it is necessary to state the condition of a network that an application may depend on. A package which provides such information is e.g. the Network Weather Service [91], which periodically monitors and dynamically forecasts the network conditions like bandwidth. Fricke [41] works on the normalization of the measurements taken by different network monitors and expressing this data in a Grid Object *Network Profile*, as defined in Section 5.8.3. The information is e.g. used to qualify the accessibility of a site *before* a large file transfer is initiated. It allows a migration service to select or rule out potential migration hosts based on their network connectivity.

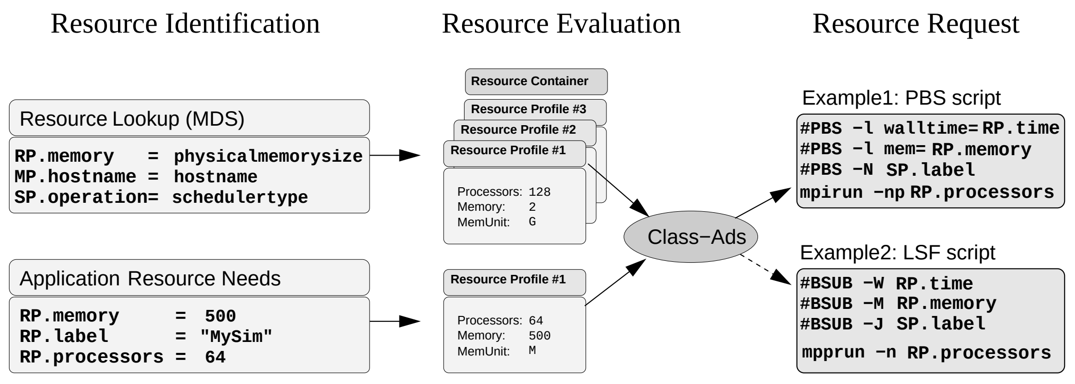


Figure 5.10: Resource profile (RP), service profile (SP) and machine profile (MP) translate between different application vocabularies. An MDS server is queried for queue properties, while an application identifies its resource needs. Constraints and requirements are expressed as Grid Objects. Class-Ad matching filters an appropriate queue and machine. Jobs are submitted, which requires the mapping to e.g. PBS, LSF or Globus RSL directives.

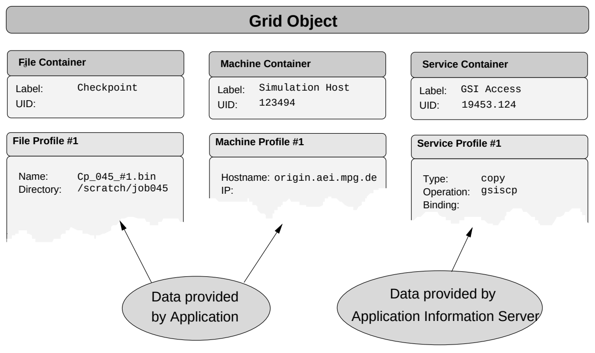


Figure 5.11: An application provides information on file, e.g. a checkpoint file and it may know about the local machine as well. If the application has no knowledge on access methods, the Application Information Server (or other database) can be used to complete the Grid Object description on how the file can be accessed.

Time: Future work includes the notion of time and time intervals. The present system of Grid Object profiles is static in the sense that profiles do not timeout or become invalid. For example defining resource co-allocation and advanced reservation requires the use of time-out constructs to describe the beginning and expiration time of a resource. We are working on the extension of Grid Objects to accommodate these dynamic properties.

5.11 Discussion – OGSA, GODsL and GPS

OGSA defines an architecture for the Grid with aspects like security, authentication, and authorization. Grid Peer Services define the notion of a redundant service environment for Grids. GODsL allows for a compact description of Grid content. In this section, we compare the aims of OGSA [37] to the GPS and GODsL. We give an answer to the question, in how far OGSA overlaps with GPS and GODsL. We show in fact that the three packages do not rival each other.

We consider OGSA as an important approach to realign the different Grid technologies that are developed in independent projects and allow them to interoperate. Based on the experiences with distributed service environments, we believe that there are two crucial points for the success of this architecture:

- **Simplicity:** OGSA should be a *lightweight* and *modular* specification. A monolithic, all-in-one specification will likely defeat the purpose of service compatibility as well as community acceptance. Some of the OGSA concepts like service factoring are valuable, and ambitious, as we have seen in our service monitor (Section 9.4.3), which creates and maintains service and application instances.

- **Application Base:** User acceptance can only be reached by having real-world applications as the driving force for the development of any technology. OGSA definitely facilitates the interoperability between Grid technologies. Its success to interoperate with customer application will depend on how soon user applications are involved.

OGSA vs. GPS: There is a simple reason why GODsL and GPS implementations were not able to conform to the OGSA specification: By the time that OGSA was released, most of the design work for GPS and GODsL were completed and first experiments were shown [50]. The OGSA code base was – at the time of the writing – still in a stabilizing phase as sample implementations of OGSA conformal services became available.

OGSA and GPS both use web services as a mean to communicate between independent applications. OGSA aims to define an "architecture" for the Grid and stresses aspects like security, authentication, and authorization. These features have not been the focus in the Grid Peer Services, which concentrates on a redundant service environment.

The difference between GPS and OGSA are more on the level of *service topology*: Grid Peer Services promotes a fault tolerant service topology by using peer-to-peer strategies. With the migration service environment (chapters 7 and 8) we show that a P2P approach allows for reliable services on top of a unreliable Grid infrastructure. As more high-level services are developed for *global* Grids, designers are faced with the same problem. Likely they will reach a similar design decision. The requirements of fault tolerance in a Grid environment have to be solved by any OGSA based application. We believe that the peer-to-peer approach is a viable concept.

Besides adding security and authentication, OGSA offers many features like the *service factoring* to generate a service instance. GPS can make immediate use of such technology, e.g. in the "*service monitor*" that we use to manage and supervise user codes and service applications. In this respect, we are eager to enhance our code with sophisticated OGSA techniques.

OGSA vs. GODsL: The individual profiles of a Grid Object fall short in respect to the capabilities of specialized tools. For example,WSDL and OGSA provide a far more sophisticated web service description than a service profile in a Grid Object. Nevertheless, both mechanism are conceptually different: GODsL describes content and correlates various aspects of a generic entity into a single handle. WSDL focuses on the description of an isolated aspect like web services description. While it is possible to stretch the concept of WSDL accommodate other information as well, WSDL remains a system for services.

When analyzing the advanced Grid scenarios in the introductory chapter, we came to the conclusion that only focusing on services itself is an inefficient abbreviation of the full problem scope. The isolated description of single aspect of a Grid object (like it's resource aspect, machine location or service methods) is inadequate to capture the abstraction process that is necessary for advanced Grid scenarios. In this thesis, we have explored the capabilities of such an encompassing view on entities on a Grid.

However, to link up with the future development of web services, the standardized description of web service in a Grid Object's service profile is important. The OGSA code base has been evolving over time and has now reached a stage where it starts to stabilize. For a future migration service infrastructure, which conforms to the OGSA specification, we have to make sure that GODsL service description is compatible with OGSA based services description.

With GODsL we have also addressed the problem of characterizing legacy objects on a Grid, which are not web-service compliant – and never will be. These packages pose the majority of Grid software infrastructure today. Any application-centric service infrastructure which intends to interoperate with legacy systems, must come with a description scheme like GODsL to describe capabilities. Providing web service wrappers to the legacy code is an option but requires the modification of installed software. It remains to be seen, if OGSA based services will deal with proprietary applications directly or if OGSA environments will only incorporate web service conformal codes. We chose to include ubiquitous legacy middleware in its current form without requiring additional service wrappers.

Synergies: The migration service environment with its GPS topology is a loosely coupled set of services, whose underlying service communication is so modular that it can be easily replaced, e.g. with a OGSA conformal messaging system. The OGSA specification contributes authentication and security aspects, which have been of minor focus in this version of the service environment. OGSA with SOAP, as the underlying request protocol in OGSA, allows for sophisticated security arrangements in our environment, e.g. multi-level encrypted request documents which would allow the secure request relaying into firewalled domains. It would be a intriguing project to switch the compound migration and spawn services to OGSA conformal communication, thereby taking advantage of OGSA features like security features and service factoring. In Section 8.4, we discuss security issues which are specific to a migration environment: they result from the alternation between execution and file state.

Chapter 6

The Request Handler

Failure and response propagation in a distributed service environment is essential to ensure fault tolerant operation. In order to experiment with propagation strategies we implemented our own request server. In this chapter we present the *Request Handler architecture*, which is the core algorithm that tracks the incoming and outgoing XML-RPC messages in all GPS applications. The request handler is implemented in the Cactus Code framework. It is used for the various Grid Peer Services, which are introduced in chapters 7 and 8.

6.1 Request Handling Requirements and Strategies:

The request handler manages incoming and outgoing messages and is a middle layer located between the HTTP communication layer and the application which provides or requests services. The handler is the first thing that an incoming message sees, after it has been received by the HTTP layer. The server system design focuses on providing the following properties:

- **Request Reception**: The handler accepts incoming RPCs and tracks their process. It identifies and authenticates the originator of the message.
- **Procedure Calls**: The handler must execute the requested RPC and determine the result of such a RPC. It must be able to distinguish between faulty and successful calls.
- **Request Reply**: The request handler returns information which was generated by the remote procedure call to the original requestor.
- **Fault Tolerance**: The handler must be fault tolerant with respect to interrupted networks or failing connections when sending messages. It has to provide proper return values when RPCs did not succeed and it must propagate this information back to the requestor.

The request handler provides basic security like password based authentication for incoming requests, but it should be noted that security has not been the focus of this work. The data flow through the different layers is shown in Figure 6.1. The individual layers add or extract specific protocol information like TCP/IP, HTTP or the GPS envelop.

Request Handling Strategies: The strategies for handling RPCs by a server can be diverse. Here we list two approaches:

- An application receives remote procedure calls and hands the incoming message over the RPC routine immediately. The RPC routine is responsible for providing the necessary fault tolerance, ensuring the authentication and completing the remote procedure call. This is a convincing approach, if only a single remote procedure call is implemented within a service application. If multiple RPC routines are hidden behind a single request handler this approach leads to unnecessary code duplication. If the RPC routines behind the handler can be exchanged, it may lead to non-uniform reply behavior.

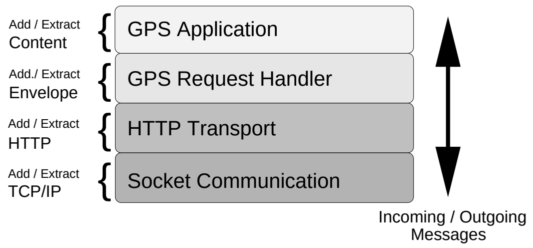


Figure 6.1: The data flow of RPC requesting and receiving applications. The GPS application at the top performs the RPC. Layer specific information is added or extracted, respectively, for outgoing and incoming messages.

- An alternative approach leaves the RPC service routine unburdened with any request bureaucracy. Instead the request handler is responsible for ensuring that a message is well-formed that the client is authorized to pose a request and that the result is returned in a fault tolerant manner. The RPC routine is called by the handler. The RPC's feedback is returned to the request handler, which propagates it back to the client. The request handler is in charge of the successful transfer of the reply data.

We chose the latter approach, because it leaves the service procedure unburdened with any request management and keeps the request tracking overhead with the request handler. It allows a programmer to write very lean RPC routines and take advantage of already existing message management in the request handler. We can replace, add and remove the RPC routines while relying on the message transfer of the request handler backbone.

6.2 Request Handling within the Cactus Code Framework

The request handler architecture is implemented within the Cactus Code Framework and uses the XML-RPC-epi[1] library. xmlrpc-epi was chosen over other implementations, because it separates the transport from the encoding aspect and yields a very compact xml-rpc parser. The Cactus Framework (cf. 3.4) was chosen for several reasons:

- *Architecture Independence*: Cactus provides an architecture independent framework that allows easy portability of applications to a variety of platforms. As discussed in the chapter on Grid characteristics, this is an important issue, when a Grid service is deployed in a heterogeneous environment.

- *Modularity*: The Cactus thorn concept matches the request handler architecture: The request handler is realized through a specific thorn, while the different RPC capabilities can be moved into a separate thorns. A service executable is generated by combining the request handler thorn with the desired RPC thorns.

- *User and Application Base*: The Cactus Code framework is used by a number of large-scale numerical projects. Since the request handler can also be used by Cactus based simulations, it allows us to experiment with real world programs, rather than using the ubiquitous ray tracing and prime number searches as test applications.

[1]`http://xmlrpc-epi.sourceforge.net`

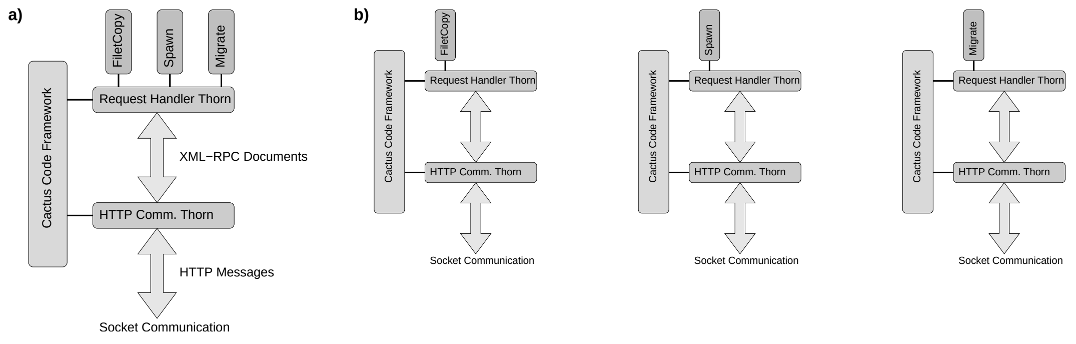


Figure 6.2: The request handler is located between the request communication layer of the Cactus HTTP thorn and the RPC layer, which provides the various remote procedure calls. The RPC thorns register their methods with the request handler. Any number of RPC thorns can be attached to the request handler.

The left diagram in figure 6.2 illustrates how different thorns are used to generate a single executable with multiple Grid services: three thorns, which provide file transfer, migration and spawning capabilities are attached to the request handler thorn. The HTTP Communication thorns provides basic socket communication. It receives incoming messages and does little else than passing them on to the request handler. The request handler supervises incoming and outgoing request. The thorns, which contain the RPC methods, are located on the top level and register their methods with the handler. These thorns focus purely on performing the RPC service. The three diagrams to the right figure 6.2 demonstrate the flexibility of the thorn concept: a single executable for each of the three service methods can be easily generated.

6.3 Request Handling - Operation Modes

A request which is communicated between a requestor and a provider can be classified in two operation modes. What mode is chosen is usually dependent on the situation and can be set as part of the transmitted request:

1. **Request - Response**: A client requests a service and a reply is returned by the server. The reply information can be anything from a database entry to an error code.

2. **Notification**: The server sends unsolicited information to clients. In accordance with the Peer-to-Peer model, servers can act as clients and vice versa. A notification is done without having received a request prior to the send. E.g. a notification mode is chosen when applications are *pinged* to determine their runtime status. Notification mode is also used when information is *broadcasted* to a range of application without requesting feedback to cut down the protocol overhead on the receiving side: an application can inform its peers that it is about to shutdown.

6.4 Request Envelope

A fault tolerant operation mode requires the back-propagation of return codes. The necessary address information has to be gained as early as possible. For example, if a request handler is not responsible for extracting the information on who sent the request, but leaves the RPC code in charge of this, it cannot provide feedback to a client if the requested RPC does not exist.

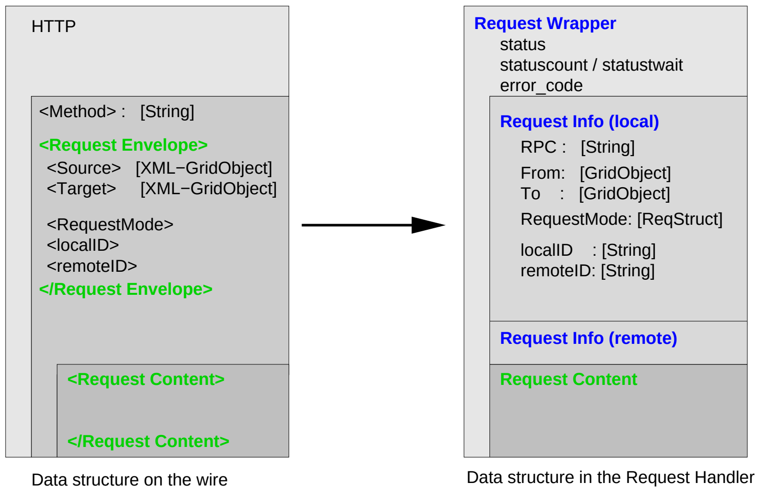


Figure 6.3: A request on the wire is expressed as an XML-RPC document, wrapped by HTTP. The request body is further structured into a *Request Envelope* and the *Request Content*. Upon arrival, the request handler stores the data in a request wrapper structure with additional data like state information.

A generic XML-RPC message is distinguished in two parts: the *method*, which contains the name of the requested operation and the *data* which is associated with that call. Our implementation of the request handler demands that the data which accompanies the RPC can be further divided into the *Request Envelope* and the *Request Content*. These sub-sections in the XML document are tagged `<RequestEnvelope>`, `</RequestEnvelope>` and `<RequestContent>`, `</RequestContent>`.

The request envelope contains all information necessary to send a message to the request originator. The request envelope provides information, which allows us to reply to the client in an early stage of the full RPC execution chain, even if the RPC itself failed or if the request content is not well formed. The request envelope contains information on the source and target of the request, expressed as Grid Objects. In the following sections we describe envelope in more detail:

6.4.1 Request Source

This sub-structure in part of the request envelope and describes the object, which sends either a request or reply to a request. In particularly it contains:

- **Grid Object:** describes the originating application or service through a machine profile and service profile. The service profile defines how to provide possible feedback (e.g. through HTTP, with a specified port, RPC binding and methods).

- **Authentication Information:** a structure which holds public keys, passwords, etc. The current version of the request handler features only password authentication.

6.4.2 Request Target

This sub-structure of the request envelope describes the application or service, which is intended to receive and execute the request. It consists of the following data:

- **Grid Object:** describes the requested service. The host information is provided by a machine profile, the service contact is characterized through a service profile.

- **Authentication Information** a structure which contains data to identify the provider as trusted service.

6.4.3 Request Properties

The request property structure captures the information, which is needed to process the request in the request handler. We briefly discuss its content.

Request ID: The request ID identifies a request document, which communicated between two peers. For the duration of a single message exchange, the participants can be categorized in source and target peers -independently whether they act as service clients or providers. Each peers assigns an the request message an individual ID:

- **Source ID:** This ID is assigned by the sending application or service. It is used to distinguish multiple requests. A Source ID is mandatory.
- **Target ID:** This ID is assigned by the receiving peer. At the beginning of a message exchange the target ID is not set. The target ID is assigned by the receiver as soon as the request arrives in the handler.

Reply Modes: The reply mode defines at what stage during the execution of a reply, feedback is sent to the client. Possible reply modes are:

- **Reply on receive**: A reply is generated as soon as the request is received and pooled.
- **Reply on method start**: A reply is generated when the execution of the remote procedure call is initiated.
- **Reply on method end**: A reply is generated when the execution of the remote procedure is finished.
- **Reply result**: A reply is generated which contains the result of the RPC. If the RPC does not provide an error code, the success or failure of the RPC execution is transmitted.
- **Reply never**: No reply is generated. This is the default mode.

Two-Phase Requests: The reply-on-receive mode allows for two-phase requests of web services. Imagine a client requesting an service, which he wants to cancel or pause during execution. The client needs to receive a handle to the web service operation *before* the copy procedure starts. Reply on receive can perform exactly this task.

6.5 Request Content

While the envelope section is standardized with respect to the information on the sending and receiving application, the request content is interpreted by the actual RPC method only. Its format is determined by the remote procedure call. The request content structure in the XML message is a `<RequestContent>,</RequestContent>` tag. The request handler extracts this sub-structure and passes it on to the remote procedure call.

6.6 Request Handling States

In this section, we introduce the concept of a *request communication channel* to abbreviate request overhead in repeated massage exchanges between peers. We continue with a detailed discussion of the request state diagram for *incoming* (Sec. 6.6.2)and *outgoing* requests (Sec. 6.6.3) and request expiration (Sec. 6.6.4).

6.6.1 Request Communication Channels

The completion of a certain task through a web service may not necessarily be achieved by a single “request and reply” pair. Multiple pairs may be exchanged before a desired result is reached. A simple example is the request for feedback as illustrated in Figure 6.4, where peer A sends a request to peer b and expects information back. B returns the information and requests feedback from A on the successful delivery of the data.

From the technical point of view, such a task can be accomplished by many independent request/reply pairs. But it can be effective to define a *request communication channel*, which stays open until the task is completed. The channel can transport multiple “request-reply” pairs which are exchanged between trusted peers until a result is reached. Such a channel allows for a single initial authentication operation followed by accelerated request operation without further authentication. This approach is e.g. used in Sun’s JXTA project [46]. The two communication endpoints in a request channel are not tied to a specific hardware, like in a TCP/IP network connection. Instead, the channel endpoints are identified with peers. This way a request channels can accompany a migrating peers, as long as they maintain their identity. In this version of the request handler, we implemented the request channels to manage multiple request exchanges between peers. We have not followed up on the idea to reduce the authentication overhead for high-frequency message exchanges.

A request-reply channel is well defined if both, target and source ID are present in a message. A missing target ID initiates a new channel, a missing source ID is treated as an error and is propagated back. Multiple request channels can be open between two peers. Each one describes a single, individual exchange, which may consist of one or more messages. A channel can be closed by a participant or by a timeout, if a reply is not received within a given interval.

6.6.2 Incoming RPC Messages

An incoming RPC messages can originate from a *client request* or arrive as a *reply* to a previously made RPC. In the first case, the message contains no target ID from previous exchanges and no request channel is open. The receiver fills the empty target ID and opens a request channel. In the second case, the receiver extracts the target ID, which he has assigned in earlier communication exchanges. Here, a channel is already existing and the pair of source ID and target ID identify the channel. Opening a channel requires the creating of channel structure. This structure contains information on the request and its state. The state diagram of a request is shown in Figure 6.5. The channel structure is added to the *request pool*, which tracks the different request channels. The specific operations, which are carried out by the request handler for incoming messages are:

1. Receiving of the serialized XML-RPC data, which is passed through from the HTTP communication layer.

2. Checking the XML-RPC code for errors and deserializing it.

3. Extracting the envelope structure, which contains information on the sender and the receiver. An error check on the completeness of the envelope data is performed.

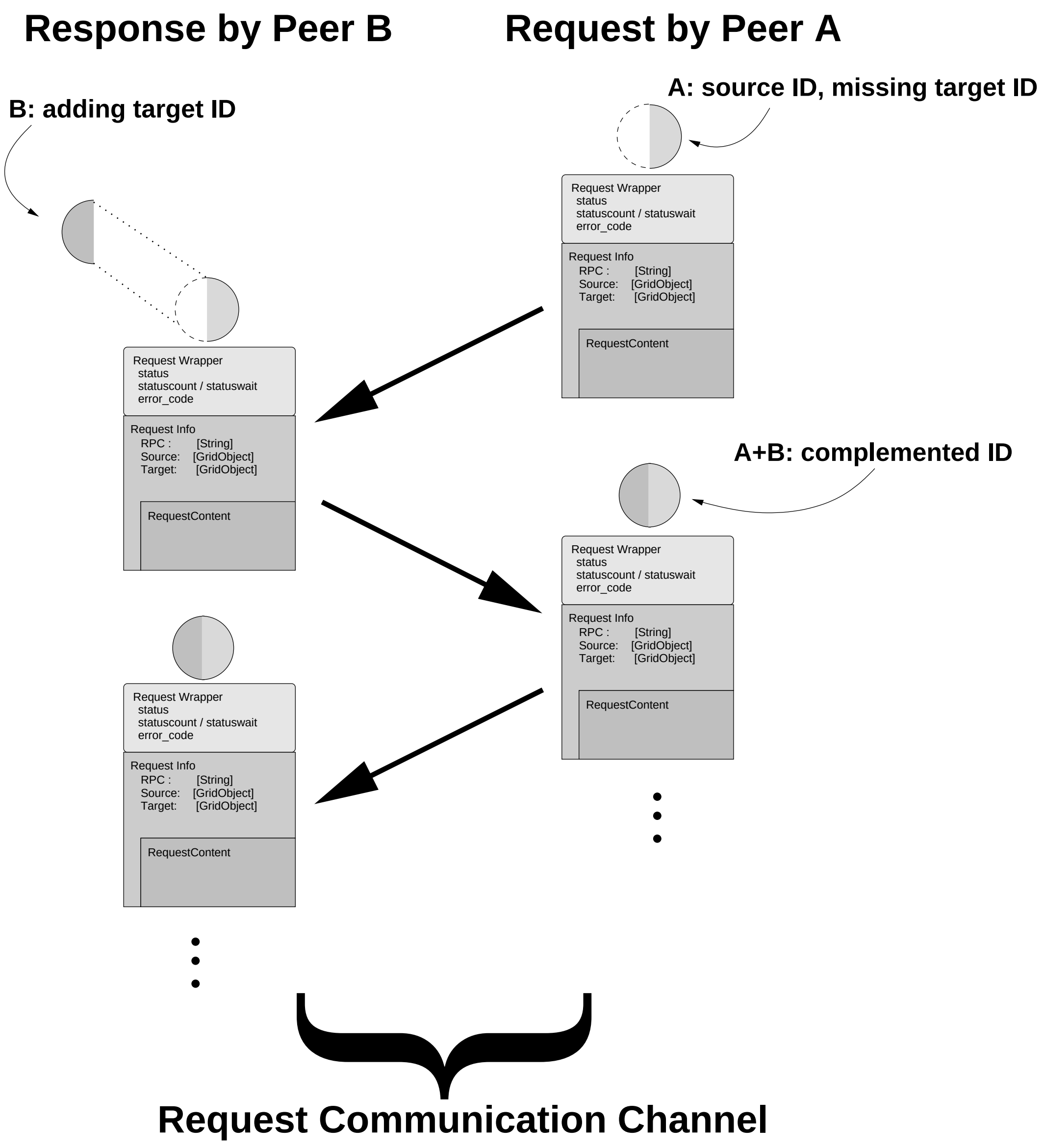


Figure 6.4: The exchange of multiple “request-reply” pairs is implemented through a *Request Communication Channel*, which stays open until the channel times out or is closed by a participant. The channel end-points are associated with peers, not with a network connection like TCP/IP. Channel end-points can “migrate”.

4. Authenticating the client. This is currently done through username/password matching.

5. Opening a request channel, if the request contains no target ID from previous exchanges. This operation consists of creating a channel structure and adding this data to the request pool.

6. Identifying the request channel, if the request specifies a target ID and update the information in the channel structure. If no channel is found reply with an error.

Incoming Message States: The incoming request passes through several states before it settles in one of the final states `REQUEST_DONE` or `REQUEST_FAILED`. An incoming message, which holds a target ID and which has been added to the request pool, receives the initial state `REQUEST_OPEN`. If the request handler comes across this state as it traverses the pool, the pending request is switched to `REQUEST_METHOD_OPEN`. The handler attempts to execute the local RPC. If the handler failed to make the call technically, e.g. if the method is not existing, the request is declared a failure: `REQUEST_METHOD_FAILED`.

If the handler was able to make the call and receives a return value other than NULL, the request is set to the state `REQUEST_METHOD_DONE`. A NULL reply is interpreted as method failure. The existence of a return value is not a statement on the success of the actual RPC. In this respect a *completed* method does not imply a successful operation. It only means that the method was completed in a controlled way. Information on the success or error of the actual RPC must be provided in the reply value. If no reply is required, the request reaches it final state `REQUEST_DONE` or `REQUEST_FAILED` depending on the outcome of the method call.

If the user has requested feedback on the result of the RPC by setting the reply mode, a reply is generated. If the RPC has returned an XML value, this value is used as the content of response. If no message was returned by the RPC a standardized message is used to build the content of an error message. The new target is the sender of the original request and the previously “incoming” request now becomes an “outgoing” request. For the reply, the handler is using the communication method, which was previously defined in the incoming request. We are currently experimenting with a primitive file communication to circumvent firewall problems on some machines: the request content is written to file and transferred to the sender’s machine. The sender checks for the existence of the file, reads it and processes it like a normal RPC, which was received through socket communication. It is a simple demonstration of how different transport methods can be used within a single request channel.

Repeated RPC Execution: In the case that a stage of the RPC execution fails, the request handler has several possibilities to handle such a failure. An unsuccessful send operation is repeated several times, until the send process succeeds or the allowed number of retrials is exceeded. Each of the *open*-states has a variable, which defines the maximum number of times that the state can be assigned to the same request. A status counter tracks the number of unsuccessful sends. If the request has no more resends available, it is set to `REQ_FAILED`, otherwise it is requeued with `REQ_SEND_OPEN`. A new resend attempt is made after a wait period. With the same motivation as in TCP/IP we use *exponential back-off* to circumvent problems of temporarily dead networks or stalled applications. If a client is requesting an unknown RPC method, it does not help to repeatedly call to this method. However, there are cases, where repeated execution can help:

- Storage space which fills up during a copy operation, but is freed over the timescale of hours.

- A service which is busy and does not receive incoming socket communication at the moment.

- A failing network connection, which is reestablished within a foreseeable time.

In general, treating failures with repeated execution makes only sense, if the problem that leads to a failure is of temporary nature. At this point, we have no means to predict the usefulness of repeated RPC executions.

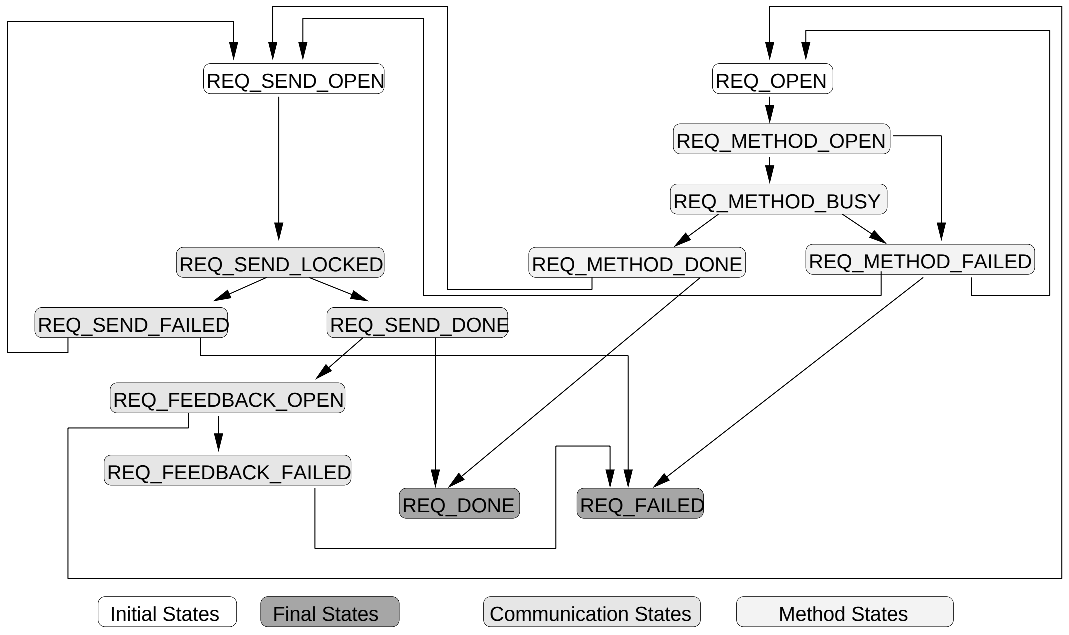


Figure 6.5: The state diagram for the request handler. Request can enter the process cycle as outgoing message with state `REQUEST_SEND_OPEN` or as an incoming request with `REQUEST_OPEN`. Final states are `REQUEST_DONE` and `REQUEST_FAILED`.

6.6.3 Outgoing RPC Messages

An outgoing message is generated either by a service request or by a reply to a previously received RPC[2]. If a request is sent out for the first time, the handler opens a new request channel. If a request *reply* is sent out, a channel was already completed with the incoming request. For outgoing messages, the handler has to do the following tasks:

1. The handler receives the instruction to send a RPC request from a application or service.
2. The handler receives a reply value from a RPC function and returns this data to the original client.
3. In both cases the handler starts or updates the channel structure, which is used to track the status of the outgoing request.
4. The handler sends the request to desired service host.
5. The handler supervises repeated sends in the case of failure.

[2]Replies are also requests

Outgoing Message States: As shown in Figure 6.5, a message which is to be sent out by the handler enters the request pool with the initial state `REQUEST_SEND_OPEN`, indicating that the message is complete and can be send. The request handler which traverses the request queue marks such an entry as `REQUEST_SEND_LOCKED` and attempt to send it. If the handler could bind to the remote peer and the send was successful, the send request is marked as `REQUEST_SEND_DONE`. If the send operation failed, the request is set to `REQUEST_SEND_FAILED` and the error counter is increased. If the error counter has not exceeded the error limit for this state, the request is requeued as `REQUEST_SEND_OPEN` with a timeout, which avoids an immediate resend. The timeout policy can e.g. follow the exponential back-off found in the TCP/IP protocol. If the error limit is exceeded, the request is marked as failed and no further attempts are made to deliver the message. A noted previously, the scheme's *send* procedure is not tied to HTTP based communication: RPC transfer through files is a crude way to circumvent restrictive firewall policies (see section. 6.6.2).

If a message is sent and a reply is expected, the request is marked as `REQUEST_FEEDBACK_OPEN` upon successful delivery. This indicates that a return message is expected and the request channel is kept open. The request stays in this state until a message arrives in this channel or until the reply wait time expires. In first case, the reply is used to updated the request pool entry: the state is switched to `REQUEST_OPEN`, the RPC method that deals with the data reply is called and a new RPC cycle is entered. If the feedback state is not resolved in time by an incoming reply, it is marked as `REQUEST_FEEDBACK_FAILED` and finally enters the `REQUEST_FAILED` state, which closes this channel. When a reply arrives to late and the request channel is already closed, an error message is returned, provided that the source is expecting feedback.

Error handling: We treat error messages are handled like normal outgoing messages. If no special RPC is known for the target of the error message, a *default RPC* method is assumed. This default request only acknowledges the arrival of a message and sets the state to `REQUEST_DONE` but does not provide any RPC operation. The request handler attempts to deliver the error messages like a normal request until all success or trials exceed the limit. Error message transmissions never solicit a reply from the target to avoid avalanches of reply messages. If a request transmission succeeded and feedback is expected (`REQUEST_FEEDBACK_OPEN`), we do not resend the request if this state times out. There is usually something seriously wrong if the receiving side gets the request but is not even able to return an error code. In such a case, we leave it to the program layer (e.g. the application) to initiate another request attempt or try out another service. The request handler itself has not enough information to draw such consequences.

6.6.4 Request Expiration

The state `REQUEST_DONE` or `REQUEST_FAILED` is the final state for all requests. If such a state is reached, the request channel is closed and the requests are no longer actively pursued. The request is not immediately deleted from the request pool, but kept in a passive mode for a certain time period. After this time, the request is deleted.

Requests with a final state cannot be deleted immediately for the following reason: an attached application may want to find out about the request which it has submitted to the request pool. It can monitor the state of the local request with the request identifier that is received. If the request is removed from the queue immediately, the application may not be able to perform such a query. For this reason, the request is left in the database in a passive mode: it can be looked up but cannot be modified. After a certain period (`REQ_DONE_WAITTIME` and `REQ_FAILED_WAITTIME`), the finalized requests are purged from the pool.

```
char *Wxml_NewRequestXML(GridObject *to,
                         XMLRPC_VALUE *xmlcontent,
                         int replymode,
                         char *method_name,
                         char *method_name_fb);

/** Example: sending a ping request **/
reqkey  = Wxml_NewRequestXML(ping_client, xcontent,
                             REPMODE_RESULT,
                             "ping", "ping_ok");
```

Listing 6.1: `Wxml_NewRequest` and `Wxml_NewRequestXML` are two routines, which request a service. Both routines specify the remote service and the local procedure, which handles a reply. The user can provide Grid Objects or an XML document (shown above) as an argument to the request.

6.7 Request Handler Design

A request channel contains information on the two participants and the properties and status of the request. The request is stored in a request wrapper and added to a *Request Pool*. This pool stores all requests with their different states. It is the request handler's responsibility to traverse the pool and treat the requests depending on their status. A request is added to the pool if a new channel is initialized; they are removed from the pool if a channel is closed or if the channel experienced a time-out.

Working and pruning the request pool is executed in regular intervals. Adding requests to the pool is event triggered, either by receiving a message from the HTTP transport layer or by a send request by the application code.

6.7.1 Request Handler API

The request handler provides a simple API for thorns to pose a request. In Listing 6.1, the prototypes for the two C routines `Wxml_NewRequest` and `Wxml_NewRequestXML` are shown, which create a request and hand it over to the request pool. Both routines specify the request target through a Grid Object, and the reply mode through an integer value. The also pass along the name of the remote procedure call which is executed *remotely* and RPC which is executed *locally* to process return information. The two routines differ in the content format, which is a Grid Object (`content`) in the first case and an arbitrary XML structure (`xmlcontent`) in the latter. Grid Objects can be used in a number of situation, for all other cases the custom XML structure is offered. Both routines return a string value, which identifies the request in the pool. It can be used to monitor the status of the *local* request. The example shows the call to make a ping request to an application, described by the Grid Objects `ping_client`. The client processes the `ping` RPC and replies with a RPC request that is processed by the `ping_ok` RPC. Listing 6.2 shows the program, which sends the ping reply to the requesting application.

6.7.2 Example RPC routines

A RPC is registered with the request handler by storing the function pointer and associate that pointer with the RPC method name. An incoming request specifies the message name and the function can be executed with the appropriate argument. In listing 6.2 we illustrate a RPC routine: the ping-RPC first extracts the request content, marked `<RequestContent>`, `</RequestContent>`. It then looks for a string value, which is marked `<pkey>`, `</pkey>`. The ping client returns this value to

```
XMLRPC_VALUE rpc_ping(XMLRPC_SERVER   server,
                      XMLRPC_REQUEST xreq,
                      void *twrap)
{
  xcontent = XMLRPC_VectorGetValueWithID(xdata,    "RequestContent"))
  xVal     = XMLRPC_VectorGetValueWithID(xcontent, "pkey");
  key      = XMLRPC_GetValueString(xVal);

  if (!key) return (NULL);
  else      return (Wxml_RetString(0,key));
}
```

Listing 6.2: `ping`: a RPC routine which replies to ping requests. The routine receives xml message, extracts the content returns the value the key-value pair.

the request handler (`return (Wxml_RetString(0,key))`). The handler propagates the data back to the application that requested the ping RPC.

6.8 Future Work

The request pool approach was chosen to permit better server performance later by threading the work on the request pool. The current request handler runs unthreaded and experiences the expected performance restriction under heavy load. In a future implementation, multiple threads operate on the request pool, yielding a higher throughput of RPC. Switching to a professional server framework should eventually be considered. Note we gain a great deal of flexibility by running the services in user space, which must be given up if Web server based request frameworks are used.

This Request Handler is intended to be used as a work bench to experiment with different error handling strategies. We found that repeated sends are essential in an application environment, where unthreaded codes simply refuse to accept RPC replies while they are busy crunching numbers. The ability to retrieve information on the various stages in a RPC execution requires too much logical overhead to be of use. While it always easy to instrument and time different phases we found it a lot more challenging to derive proper consequences of a detected behavior. It is easy to determine that a request has failed or takes considerable time. It is currently impossible to say *why* it is behaving this way and to derive alternatives.

Editing Service Operations: The current request handler does not provide an RPC interrupt system to modify service operations in progress. This feature is essential to cancel e.g. file transfer operations. To allow clients to edit ongoing service execution, the server returns an identifier to the client before the process starts. The “Reply on method start” mode of the request handler (see Sec. 6.4.3) can directly support “two phase commits” by returning the request ID. The client uses the ID to terminate, pause or resume a service process. We regard this as an important feature for web services which perform long-term operations.

Chapter 7

AIS and Fundamental Grid Peer Services

This chapter introduces the Application Information Service (AIS) and a migration service environment based on the "Grid Peers Service" concept as defined in Section 4.4. It is based on our analysis of Grid environments (Chapter 2) and motivated by the development of a fault tolerant service infrastructure (Chapter 4). As shown in Figure 1.1 on page 2, *high-level services* are built incrementally from the underlying Grid infrastructure and from *fundamental services*. The AIS provides information service capabilities to all levels. Client applications communicate with the services and AIS directly. In this chapter we start with the introduction of the AIS and fundamental services, followed by the description of complex high-level services in Chapter 8. The services are implemented as thorns in the Cactus Code framework that operate with the request handler introduced in the previous chapter.

GPS Environment: We make no special assumption on the computer hardware which supports the GPS applications. In particular, we do not restrict ourselves to homogeneous sets of machines and we make no requirements on the quality of the network, except that it is usable from time to time. If possible, we reuse existing Grid infrastructure, like Globus or batch system. We have no requirement on the Globus version, however at the time the experiments were conducted, Globus v1.4 was most the widely deployed version[1]. The GPS applications are executed in user space, they are not hidden behind a web server like Apache or Web Sphere. They do not communicate through the HTTP port 80, but in the public port range, usually found above 2000. Port numbers are not pre-defined and can be adapted to the local environment. The usermode approach was chosen to experiment with the automatic restart of service applications in the case of failure.
In this chapter we take a detailed look at the following services:

- **Application Information Server (AIS)**: The AIS acts as an information directory for the GPS and is a central part as in any distributed client/server environment. The AIS differs from the other peer services in the sense that it serves as the central registry for applications, services and files. Application can register and deregister services, files, machines etc. through Grid Objects. Applications can query the AIS for the existence, type, and status of compound objects like resource, files, etc.

- **Grid Ping Service**: A simple service construct which is inspired by the ping [74] command found on most UNIX systems. The UNIX ping allows a user to check the availability of a hardware and derive some basic information on the quality of the network interconnect. Grid Ping is a web service that intends to provide a similar functionality for applications. Applications, which feature the GPS client interface, can be pinged to check if the program is up and running.

- **Grid File Server (GFS)**: The Grid File Server provides basic file management operations on the Grid. Its services allow to copy, delete and move directories and files from one host to another. The Grid File Server interfaces with transfer methods like *secure copy (sco)* or *GSI-scp*. The GFS can as well accommodate advanced file transportation mechanism, such as *gridftp* [3] or *secure ftp (sftp)*.

[1]The latest release is Globus v2.2, Globus v2.0 is considered stable.

- **Grid Shell Server (GSS)**: The GSS executes commands on a remote machine. Similar to the GFS, it has knowledge about access protocols, like *rsh*, *ssh*, *GSI-ssh*. The GSS has the special task to start or submit programs on remote hosts. Besides execution through remote shells, Globus based submission methods through GRAM[49] (using the *globusrun* command) in conjunction with RSL scripts can be used, if they are supported on the target system.

- **Grid Resource Service (GRS)**: This service provides information on compute resources and serves as an interface to the different resource selection systems already developed. Information can be supplied manually through parameter files or resource information can be extracted from MDS databases [25]. The GRS provides an API to match resources through Class-Ads [22]. Section 5.9.6 describes, how Grid Objects are used to relate the different definition of resources.

7.1 Application Information Server

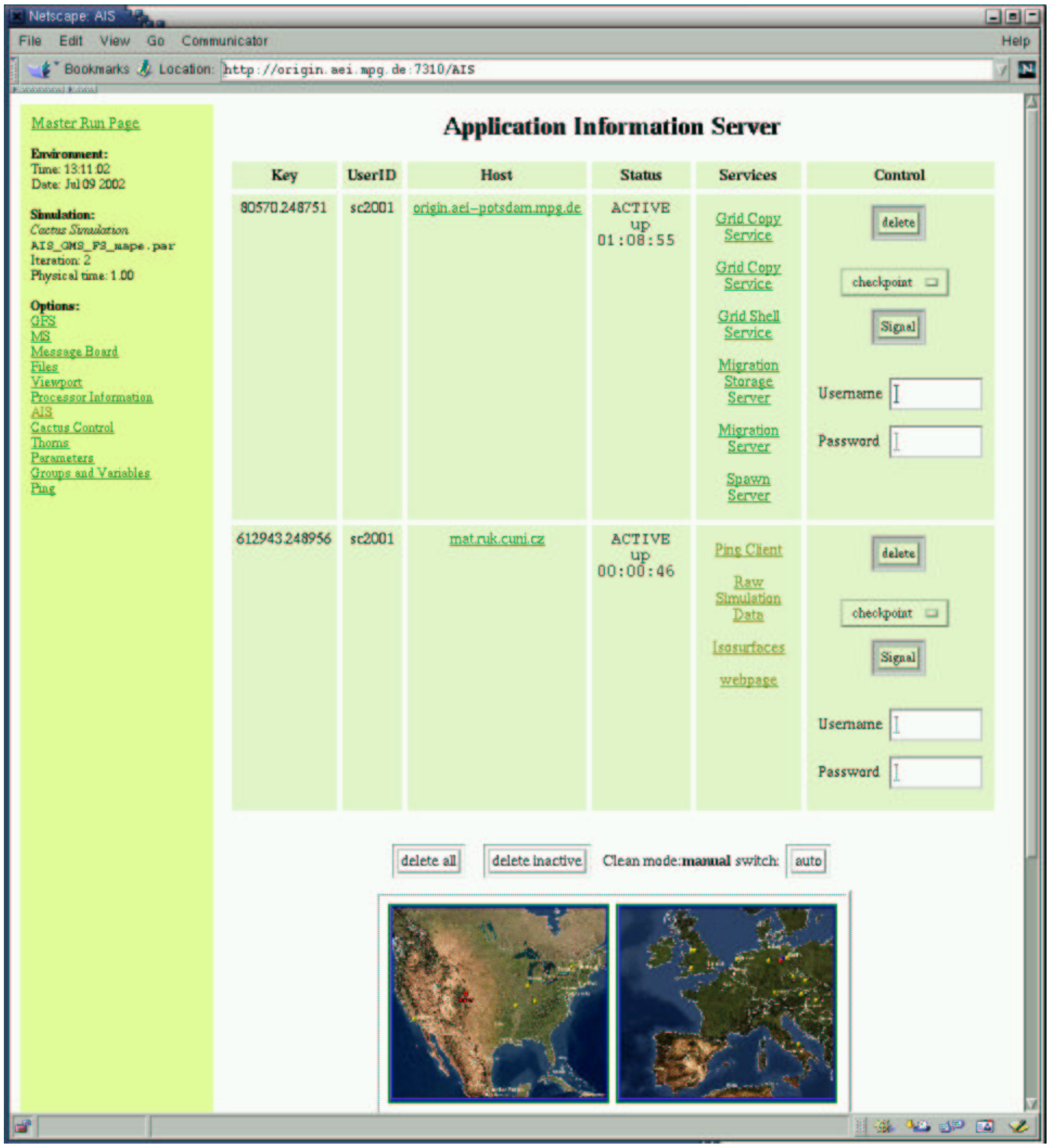
The Application Information Server (AIS) serves as the registration component in the triad of service requestor, provider and registration. It is constructed as a web service, which can be contacted by applications to query the database or deposit object information. The AIS features a simple graphical user interface, which allows a user to monitor the state of objects in the database. The AIS (unless it operates as a *personal AIS*, explained below) is designed to be contacted primarily by applications. The AIS can actively track services by using the Grid Ping. In Figure 7.1 we demonstrate the web browser interface to the AIS: the table shows two service applications, which have registered with the AIS. The top application is running on `origin.aei.mpg.de` and provides copy, shell, migration and spawn services. The second application runs on `mat.ruk.cuni.cz` and is a scientific simulation. Both applications are stored as Grid Objects. The simulation can be actively tracked, since it registers a Grid Ping client interface. The simulation announces different access methods to introspect numerical data: Isosurface streaming, raw data and the application's own web page. The AIS plots the current location of (migrating) applications in a map, shown at the bottom. The AIS can trigger a migration or checkpoint event remotely, if the client supports the appropriate RPC interface (not listed in the table). The password authentication for such signaling is shown in the right most column.

7.1.1 Primary AIS

In a distributed service environment at least one central, primary registry must exists, which resembles the root of a structured information server hierarchy. We illustrate such a hierarchy in Figure 7.2. This server must be known to all clients. All other services, including additional AIS can be derived directly or indirectly from this primary registry. The primary AIS is not necessarily a superset of all registered services but serves as the first contact point for a client.

Having a single instance of a primary AIS poses a single point of failure. This problem is intrinsic to the topology in peer networks. It can be circumvented by deploying redundant primary AIS with the same content and making these AIS known to the clients. A client learns about the redundant AIS by requesting this information through `ais_search`. The AIS may also send unsolicited update information to registered client applications. Multiple instances of the AIS must maintain a synchronized database content. If synchronization can be preserved, a client may communicate with any of the AIS. In reality, database content is not propagated fast enough to be in synchronization: While missing AIS entries can still be retrieved by a client from the other AIS, it becomes a severe problem for a client to distinguish between stale and valid data.

Deploying different types of AIS besides the primary AIS can improve the performance of AIS communication. The various AIS types may register only certain aspects: some AIS can be dedicated


Netscape: AIS
File Edit View Go Communicator Help
Bookmarks Location: http://origin.aei.mpg.de:7310/AIS

Master Run Page

Environment:
Time: 13:11:02
Date: Jul 09 2002

Simulation:
Cactus Simulation
`AIS_GMS_F3_mape.par`
Iteration: 2
Physical time: 1.00

Options:
GFS
MS
Message Board
Files
Viewport
Processor Information
AIS
Cactus Control
Thorns
Parameters
Groups and Variables
Ping

Application Information Server

Key	UserID	Host	Status	Services	Control
80570.248751	sc2001	origin.aei-potsdam.mpg.de	ACTIVE up 01:08:55	Grid Copy Service; Grid Copy Service; Grid Shell Service; Migration Storage Server; Migration Server; Spawn Server	delete; checkpoint; Signal; Username; Password
612943.248956	sc2001	mat.ruk.cuni.cz	ACTIVE up 00:00:46	Ping Client; Raw Simulation Data; Isosurfaces; webpage	delete; checkpoint; Signal; Username; Password

delete all | delete inactive | Clean mode: **manual** switch: auto

Figure 7.1: The Application Information Server (AIS) is a database which stores information on services, files or resources. The browser interface of the AIS shows several services and a user simulation. The right column shows the manual controls for checkpoint and migration signaling.

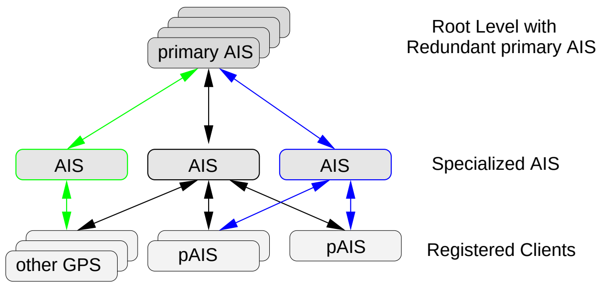


Figure 7.2: Information registries can be operated in a hierarchical structure: the primary AIS is the root server, which contains information on other AIS services. The various AIS may have different content, like selected GPS information (left), information on all services (center) or references to personal information servers only (right).

to serve fundamental services only, others monitor high-level services, store file information or track *personal AIS*. All data is stored as Grid Objects.

7.1.2 Personal AIS

The *personal AIS* (pAIS) serves as a private information server to users and reflects the scientific content of applications. For migrating applications, researchers cannot directly determine, where their application is executing or where it will relocate to next. A pAIS is used to inform the user of the current state of his simulation. The pAIS shows and visualizes data, which has been transmitted by a client application. See the genome analysis application for a demonstration of this capability in Section 9.3.

7.1.3 AIS interface

AIS interface methods are remote procedure calls, which can deposit and retrieve Grid Objects or set the state of an entry. Table 7.1 lists all RPC's that are supported by the AIS. An object is added to the database through the `ais_announce` method with an Grid Object as argument. The method is unaware of the Grid Object's content. The AIS can actively track the status by pinging those applications which have announced a ping client method to the AIS (see Section 7.2.1). Active status tracking allows the AIS to detect hanging programs or dead network connections, which render a remote service useless: It e.g. permits the AIS to verify the availability of a Grid Object (firewalled ?) by pinging it before publishing the information.

A Grid Object can be set *active/inactive* with the `ais_setstatus` method, independently of whether the object describes a file, a service, a resource or collection of those. An migrating application e.g. declares its status *inactive* prior to migration and resets it to *active* after recovering on a new host. Machine entries can be inactivated as they are shutdown or lose network connectivity; resource entries which describe queue properties can be inactivated when the queues are turned off. Simple *key-value pairs* can be added and retrieved to the database through the `ais_info`, `ais_getinfo` method. An example for a key-value entry is the current iteration of an application.

The interface methods (Table 7.1) satisfy the requirements of the migration and spawn scenario. Other scenarios may require additional database methods.

AIS method	Arguments	Description
`ais_announce`	*Grid Object*	Registration of a Grid Object with the AIS database.
`ais_update`	*Grid Object*	Updating of a Grid Object in the AIS database.
`ais_destroy`	*Grid Object*	Remove the Grid Object in the AIS database.
`ais_setstate`	*Grid Object*, *integer*	Set the application to a different state (e.g. declare inactive, inoperational, active).
`ais_info`	*key, info string*	Depositing information by simple key/value pairs.
`ais_info2file`	*info string, MIME Type*	Append information string to a file and announce the file with the specified suffix and MIME Type.
`ais_search`	in: *Grid Object*, *count* out: *Grid Objects*	Returns the number of Grid Objects, which match the information of input argument. E.g. used to search for a specific *service type*.
`ais_get`	in: *ID* out: *Grid Objects*	Returns the specified Grid Objects.
`ais_getstate`	in: *ID* out: *state*	Returns the status of an AIS entry. This method is e.g. used to verify the activity of services or validate file profiles.
`ais_getinfo`	in: *ID* out: *key-value*	Returns a specified key-value pairs.

Table 7.1: Application Information Server: web service interface to deposit and retrieve Grid Objects.

Ping method	Arguments	Description
`ping`	in: *ID* out: *ID*	Client interface: receives a ping-ID and returns the transmitted ID.
`ping_ok`	in: *ID*	Server interfaces, which receives the returned ping request, times and evaluates the ping response.
`ping_rec`	in: *Grid Object*	Server interface, which receives the applications to ping as a results of a `ais_search` request to the AIS.

Table 7.2: Grid Ping: Web Service Interface.

7.1.4 Related Information Directories

Information services exist in any distributed environments. We have discussed several solutions in Chapter 3, like Globus MDS, UDDI and WSDL. The data Grid community is using file replica manager in their distributed file environment to monitor the data files (and its copies) of high-energy physics experiments. Our AIS differs from the described system in the respect that it primarily stores Grid Objects as a “neutral” format, but with arbitrary content. It is not intended to store an isolated aspect, like services or resources. The AIS acts like a data warehouse and allows services, applications and users to deposit and retrieve information. The information format is defined (Grid Objects), while the content of those objects is arbitrary and only relevant to the clients.

7.2 Grid Ping Service

The Grid Ping Service adds a “heart-beat” client to an application, which is similar to the ping program found on most UNIX systems. The Grid Ping Services provides a similar functionality to applications. A Grid Ping Server sends a ping request to an application, which has the Ping client service enabled. The client replies with a return request indicating that it has received the message. The Grid Ping Server uses this information to determine whether an application or another service is available in the sense that it is running *and* is reachable by network. Currently it cannot distinguish between a failing machine, network or application. A future version distinguishes application errors from a machine or network failures by proper interpretation of socket error codes. Note that using the UNIX ping is problematic since ICMP packets are frequently filtered.

7.2.1 Ping Interface

Ping client and server provide an individual RPC, listed in Table 7.2. The server sends a ping request to a client interface and the client RPC returns a reply. (The example RPC code was shown in Section 6.7.2).

7.2.2 Application Monitoring and Firewall Detection

The Application Information Server can be set up to use Ping Services in order determine whether the registered services and applications can be contacted and are open to RPC calls. The temporal communication graph 7.3 shows this operation mode: A client registers its ping interface with the AIS (`ais_announce`) . The Ping Server queries (`ais_search`) the AIS for these ping client interfaces and starts to ping the applications (`ping`). The Ping Server informs the AIS about an *active* state on success or an *inoperational* state for multiple ping failures. (`ais_setstate`). The AIS updates its database accordingly. The ping server can be used in various configurations, e.g. as part of the automatic recovery shown in Section 9.4.4. In the spirit of Peer-To-Peer, the ping server can also be used by applications to check for the availability of any other service, like an AIS.

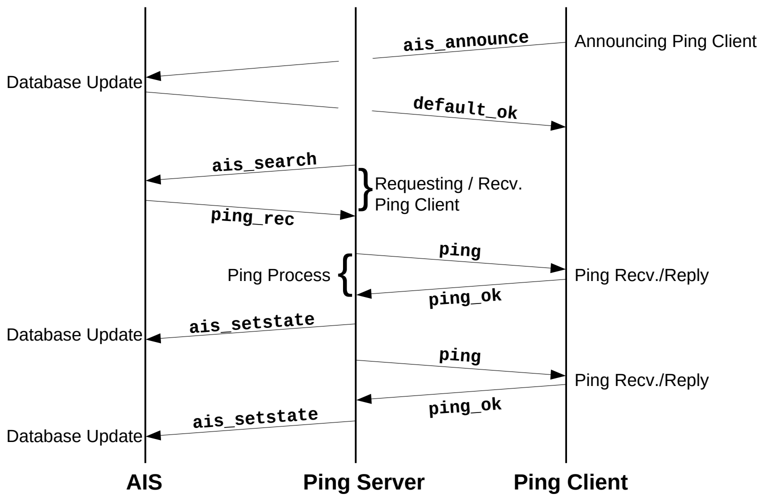


Figure 7.3: The Grid Ping Service determines the accessibility of an application or service instance. An application may become unaccessible through machine or software failure or through network problems. The AIS utilizes the ping server to actively track the status of services or user applications.

Firewall Detection: The Ping Server can be used to check for the existence of firewalls and scan for open ports. For a firewalled application it is not easy to detect whether ports are accessible by outside applications. An application can detect this by acting as a *Ping Server* and sending `ping` requests to a persistent client, e.g. a primary AIS. If the AIS' replies fail to get back, the application can assume that it is firewalled and take appropriate measures: it can inform the AIS that active tracking won't work but that the it will update itself to the AIS more frequently to signal activity.

7.3 Grid File Server

The Grid File Server (GFS) offers fundamental file management like copy, move and delete operations on a file, a collection of files or directories. The GFS does not provide these operations itself, but uses the existing infrastructure (as shown in Figure 1.1, on page 2). In Figure 7.4 we show the process of retrieving a GFS from the AIS and requesting a file copy operation. The arguments of the service requests are Grid File Objects, which specify e.g. the source and target files for copy operations, or the target files for deletions. The GFS retrieves information about the supported access methods from the AIS.

7.3.1 GFS Interface

Table 7.3 lists the operations, which are used by a migration and spawn service. Arguments to the GFS interface are Grid Objects. The GFS measures the file transfer rates for copy operations as shown in Figure 7.5. We intend to make this data part of the selection process for machines: in future we

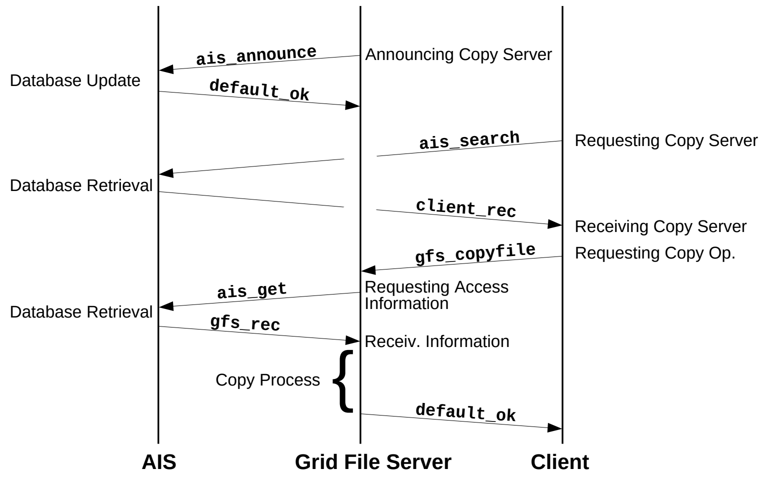


Figure 7.4: The Grid File Server (GFS) provides various file management methods to clients. After registering with an AIS, a client application requests the File Server from the AIS. The client sends a copy request to the GFS. If the GFS is not able to determine the access methods for source and target file, it queries the AIS for this information. The return value of a file operation is sent back to the client as a default reply.

GFS method	Arguments	Description
`gfs_copyfile`	in: *Source GFO, Target GFO* out: *error code*	file, directory copy operation
`gfs_movefile`	in: *Source GFO, Target GFO* out: *error code*	file, directory move operation
`gfs_delfile`	in: *Grid File Object*	file, directory delete operation
`gfs_rec`	in: *Grid File Object*	receiving AIS data

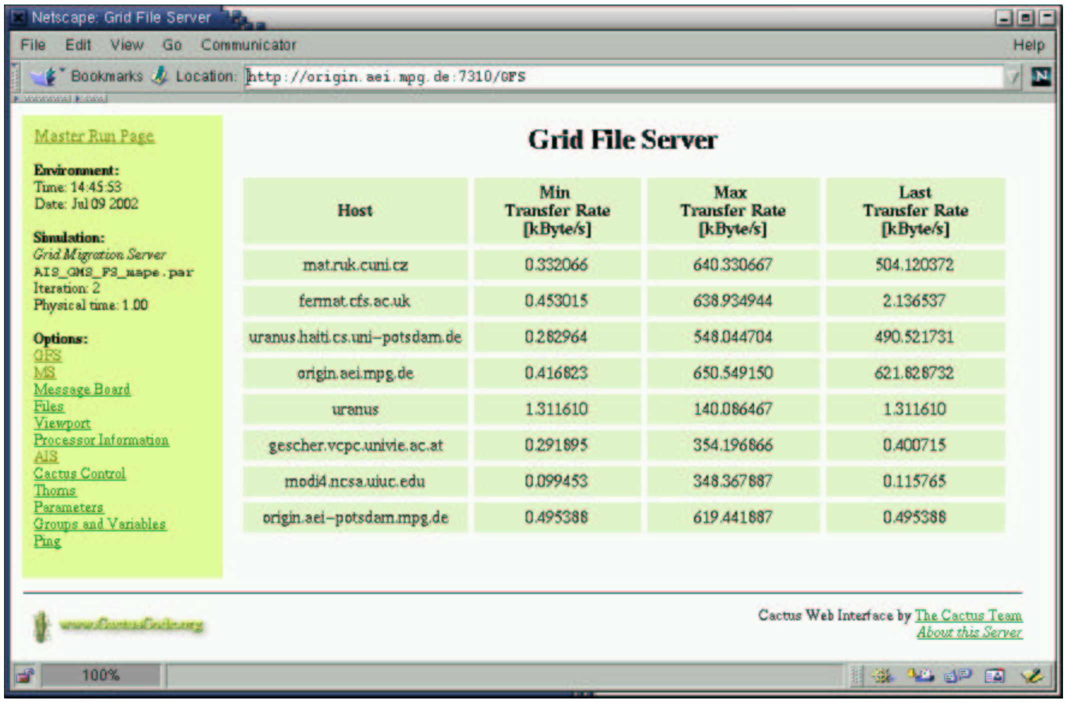
Table 7.3: Grid File Server: Web Service Interface

want to ignore hosts with inferior network that cannot be supplied with data files in an acceptable time.

7.3.2 Supported Infrastructure

The GFS itself does not copy files, but uses the techniques that are available on the source or target machine. Source and target machines do not need to support the same transfer methods, since copy operations are carried out in a two-way process. The information on the methods is either part of the copy arguments (service profile in a Grid Objects) or is retrieved from an AIS. Grid File Services support copy and move methods with the syntax for *remote copy* (rcp), *secure copy* (scp), secure file transfer protocol (sftp) and *Globus Secure Infrastructure-copy* (GSI-scp). Delete operations are based on shell access which uses remote shell (rsh), secure shell (ssh) and GSI-ssh.

Authentication: Depending on the shell and file operation, authentication is achieved through public/private key authentication for ssh/scp. For Globus based access a Globus proxy is used. We introduced the Globus Security Infrastructure in Section 3.1.1.



http://origin.aei.mpg.de:7310/GFS

Master Run Page

Environment:
Time: 14:45:53
Date: Jul 09 2002

Simulation:
Grid Migration Server
`AIS_GMS_FS_mape.par`
Iteration: 2
Physical time: 1.00

Options:
GFS
MS
Message Board
Files
Viewport
Processor Information
AIS
Cactus Control
Thorns
Parameters
Groups and Variables
Ping

Grid File Server

Host	Min Transfer Rate [kByte/s]	Max Transfer Rate [kByte/s]	Last Transfer Rate [kByte/s]
mat.ruk.cuni.cz	0.332066	640.330667	504.120372
fermat.cfs.ac.uk	0.453015	638.934944	2.136537
uranus.haiti.cs.uni–potsdam.de	0.282964	548.044704	490.521731
origin.aei.mpg.de	0.416823	650.549150	621.828732
uranus	1.311610	140.086467	1.311610
gescher.vcpc.univie.ac.at	0.291895	354.196866	0.400715
modi4.ncsa.uiuc.edu	0.099453	348.367887	0.115765
origin.aei–potsdam.mpg.de	0.495388	619.441887	0.495388

Cactus Web Interface by The Cactus Team
About this Server

Figure 7.5: The Grid File Server measures transfer rates. In a future we want to make this bandwidth data a part of the resource selection process.

Future Infrastructure: GridFTP[2] is a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. The GridFTP protocol extends the traditional FTP with features like GSI authentication, parallel streams and third-party transfers. In conjunction with *Reliable File Transfer (RFT)*[3] GridFTP achieves a high-degree of fault-tolerance on both client and server side. We are looking for exactly this kind of service to stage migration files of significant size. Interfacing with GridFTP promises a far more efficient file transfer than with the rcp and scp operations. Since the underlying copy operations are abstracted from the GFS service interface, such additional copy operations can be quickly added. The GFS will also be able to support the two-phase commit of copy requests (sec. 6.4.3).

7.4 Grid Shell Service

The Grid Shell Service (GSS) provides simple shell access to remote hosts and executes the requested commands on behalf of the user. The target machine is specified as a Grid Object and may contain profiles that describe the supported access modes. If no such information is specified, the GSS queries the AIS for Service Profiles, which define the access types.

7.4.1 GSS Interface

Table 7.4 lists the operations of the GSS. Like the GFS, the GSS relies on existing services to access a site. We are currently using rsh, ssh and GSI-ssh based access methods. If available, we use Globus

[2] `http://www.globus.org/datagrid/gridftp.html`
[3] `http://www-unix.mcs.anl.gov/~madduri/RFT.html`

GSS method	Arguments	Description
`gss_shellcmd`	in: *Command String*, *Grid Object* out: *error code*	Argument specifies access method and target machine, method executes a command on the specified host.
`gss_submit`	in: *Grid Objects* out: *error code*	Submits a job to the specified host(s). Grid Object specifies submission system, target machine, resource requirements and start up sequence. Multiple Grid Objects are used for a meta-computing execution.

Table 7.4: Grid Shell Server: Web Service Interface.

GRAM services (globusrun) to launch applications (see below).

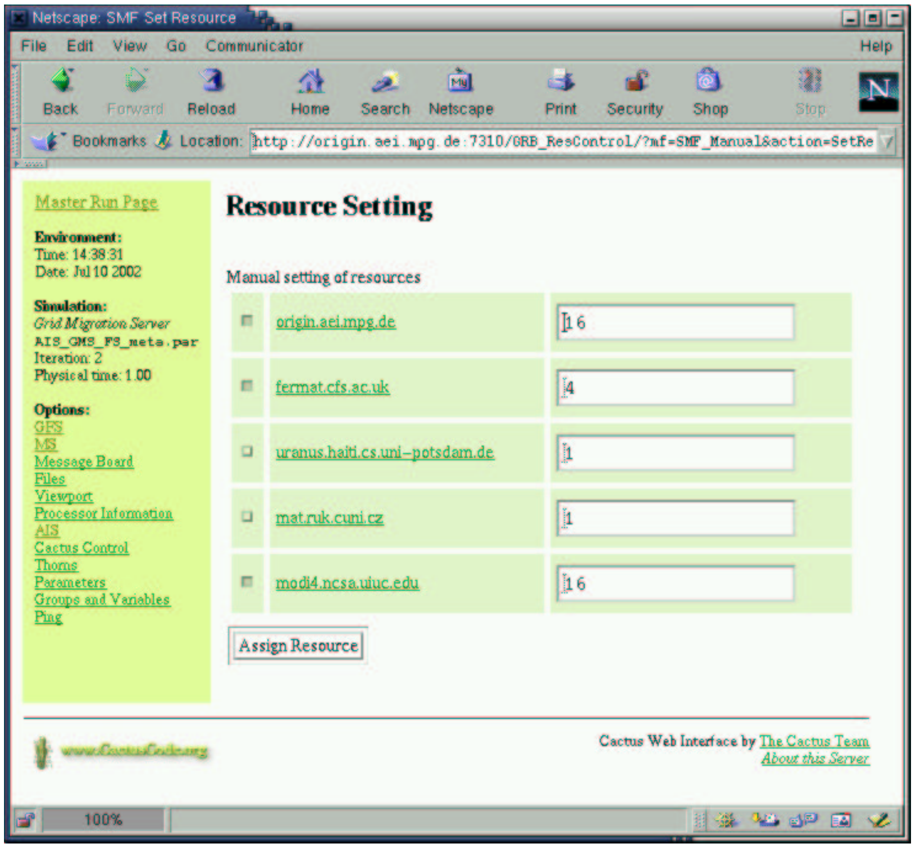
7.4.2 Job Submission

In addition to simple shell access, the Grid Shell Service starts applications on remote hosts. The GSS does not need to run on these machines, as long as it can interface with them. Job submission is possible through traditional batch submission systems or Globus GRAM services. These submission interfaces usually require the specification of resources such as memory consumption, runtime, etc. The Grid Object must contain a Resource Profile to details the resource characteristics of the application and to fill out batch scripts. The execution command is specified in the resource profile's run command attribute. The GSS has a simple interface system to register modules which generate the batch scripts for the various systems. If the GSS cannot submit a job, it returns an error code to the requesting application.

Figure 7.6 shows the resource selection through a web form of the Grid Resource Service, discussed further down. The chosen resources are used to perform a meta-computation across three hosts. The Grid Shell Service recognizes the three machines and their Globus service and generates the appropriate RSL script. Below, we show the RSL script, which was assembled by the resource selection shown in screenshot in Figure 7.6:

```
+
(* origin.aei.mpg.de *)
(&(resourceManagerContact="origin.aei.mpg.de")
   (count=16)
   (jobtype=mpi)
   (label=414935.813973_0)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 0))
   (directory=/data/sc2001/WORM2)
   (executable=/data/sc2001/WEXESERVER/cactus_w2-32)
   (arguments=/data/sc2001/WORM2/WormNG4F_man_W414935.813973-856.par)
   (stdout=/data/sc2001/WORM2/LOG_W414935.813973.log)
   (stderr=/data/sc2001/WORM2/LOG_W414935.813973.err)
)

(* fermat.cfs.ac.uk *)
(&(resourceManagerContact="fermat.cfs.ac.uk")
   (count=4)
   (jobtype=mpi)
   (label=414935.813973_1)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 1))
   (LD_LIBRARYN32_PATH
        /opt/scsl/scsl/usr/lib32/mips4:/opt/scsl/scsl/usr/lib32:
        /opt/mpt/mpt/usr/lib32/mips4:/opt/mpt/mpt/usr/lib32:/opt/MIPSpro
        /MIPSpro/usr/lib32/mips4:/opt/MIPSpro/MIPSpro/usr/lib32)
```


Netscape: SMF Set Resource
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://origin.aei.mpg.de:7310/GRB_ResControl/?mf=SMF_Manual&action=SetRe
100%

Master Run Page

Environment:
Time: 14:38:31
Date: Jul 10 2002

Simulation:
Grid Migration Server
`AIS_GMS_FS_meta.par`
Iteration: 2
Physical time: 1.00

Options:
GFS
MS
Message Board
Files
Viewport
Processor Information
AIS
Cactus Control
Thorns
Parameters
Groups and Variables
Ping

Resource Setting

Manual setting of resources

■	origin.aei.mpg.de	16
■	fermat.cfs.ac.uk	4
☐	uranus.haiti.cs.uni-potsdam.de	1
☐	mat.ruk.cuni.cz	1
■	modi4.ncsa.uiuc.edu	16

Assign Resource

www.CactusCode.org

Cactus Web Interface by The Cactus Team
About this Server

Figure 7.6: Grid Shell Service: the GSS supports various ways to launch applications. It is able to launch a meta-computing jobs by composing a Globus RSL script and submitting it. This resource configuration is composed manually in the resource service. It launches a 36 processor simulation, of which 16 processors are used on `origin.aei.mpg.de`, 16 processors on `modi4.ncsa.uiuc.edu` and 4 processors on `fermat.cfs.ac.uk`.

```
   (LD_LIBRARY64_PATH
        /opt/scsl/scsl/usr/lib64/mips4:/opt/scsl/scsl/usr/lib64:
        /opt/mpt/mpt/usr/lib64/mips4:/opt/mpt/mpt/usr/lib64:/opt/MIPSpro
        /MIPSpro/usr/lib64/mips4:/opt/MIPSpro/MIPSpro/usr/lib64)
   (LD_LIBRARY_PATH /opt/MIPSpro/MIPSpro/usr/lib))
   (directory=/ohome10/zzallen/WORM)
   (executable=/ohome10/zzallen/EXEREP/cactus_w2-32)
   (arguments=/ohome10/zzallen/WORM/WormNG4F_man_W414935.813973-856.par)
   (stdout=/ohome10/zzallen/WORM/LOG_W414935.813973.log)
   (stderr=/ohome10/zzallen/WORM/LOG_W414935.813973.err)
)

(* modi4.ncsa.uiuc.edu *)
(&(resourceManagerContact="modi4.ncsa.uiuc.edu")
   (count=16)
   (jobtype=mpi)
   (label=414935.813973_2)
   (environment=(GLOBUS_DUROC_SUBJOB_INDEX 2))
   (directory=/u/ac/gallen/WORM)
   (executable=/u/ac/gallen/EXEREP/cactus_w2-32)
   (arguments=/u/ac/gallen/WORM/WormNG4F_man_W414935.813973-856.par)
   (stdout=/u/ac/gallen/WORM/LOG_W414935.813973.log)
   (stderr=/u/ac/gallen/WORM/LOG_W414935.813973.err)
```

Resource method	Arguments	Description
`grb_match`	in: *Grid Resource Object* out: *Grid Objects*	input defines requirements, returns all possible matches. Matching performed through Class-Ads.
`grb_add`	in: *Grid Resource Object* out: *ID*	Adds a resource to the database.
`grb_del`	in: *ID* out: *error code*	file,directory delete operation Removes resource from the database.

Table 7.5: Grid Resource Server: Web Service Interface.

)

Although the RSL syntax looks well structured and straightforward, it may contain a lot of site dependent environment settings. For this reason, the migration server's routine uses a *RSL template*. The template contains a working RSL section for each host. The migration server replaces run-specific data like executables, arguments, etc. and keeps the site specific data. This approach allows us to maintain a set of working RSL scripts and import them into the migration server.

7.5 Grid Resource Service

This section introduces the Grid Resource Service (GRS) which interfaces with existing resource databases. In Section 7.5.1 we introduce the basic layer, followed by a manual information specification in Section 7.5.2 and an interface to the Globus Meta Directory Service in Section 7.5.3. The GRS is responsible for storing information on compute capacities in a Grid, which can be individual machines or their batch queues. The GRS deposits this data in the resource profile of a Grid Objects. Resources are retrieved by the GRS web service interface.

7.5.1 Grid Resource Base

The Grid Resource Service suite consists – unlike the other services – of serveral thorns. The *Grid Resource Base* serves as a backbone that registers the interface thorns, which connect to third party resource managers. The concept is similar to the registration of service thorns with the request handler. The interface thorns can be compiled into an executable or left out depending on whether support for a specific resource look-up method is desired or not. The backbone provides the web service interface for the clients.

Grid Resource Interface: Resource Base queries the available resource systems in regular intervals. It translates the received information to Grid Objects and stores them in a data base. It provides the web service methods, as listed in Table 7.5.

Condor Class-Ads: The `grb_match` service is using the Class-Ads system to match an input Grid Object, which defines the requirement, against all machine entries in the database. Details of the matching structure and logic were explained in Chapter 5 as part of the Grid Object Description Language. The Grid Objects for machines or queues are mapped to a Class-Ad. Such a Class Ad for a single machine is shown below to the left. To the right we show the resource requirements of an application, also expressed as a Grid Object and then mapped to a Class-Ad:

```
[
  type       = "machine";
  mem        = undefined;
  procs      = 1;
  cpuload1   = 1.240000;
  cpuload5   = 1.080000;
  cpuload15  = 0.980000;
  procs      = 1;
  os         = "linux";
  qsys       = "fork";
  domain     = "aei.mpg.de";
  host       = "vidar2";
]
```

```
[
  type         = "job";
  requirements = (other.procs >=  4    &&
                  other.os    == "linux");
]
```

The machine resource example would not match this requirement Class-Ad, because it does not provide the necessary number of processors. This example from a standard MDS installation and the screen shot in Figure 7.7 highlight the problem of missing database entries, in this case, the memory attribute has no value and is undefined. The manual resource definition described in Section 7.5.2 provides back-up information in such a case.

7.5.2 Grid Resource Manual

Grid Resource Base in an interface thorn that offers the manual setting of resources. It parses a list of resources which is provided by the user. It serves in the following ways:

1. It gives the user a simple way of defining a small set of machines. From our own experience, the number of machines can be small if the simulation requires specialized compute resources. The manual setting allows a user to define such a small pool. without the overhead of third party resource monitoring programs.

2. It serves as a backup information repository. In some cases we experienced missing data in the installations of resource systems like MDS [25]. The manual completion of data solves this problem.

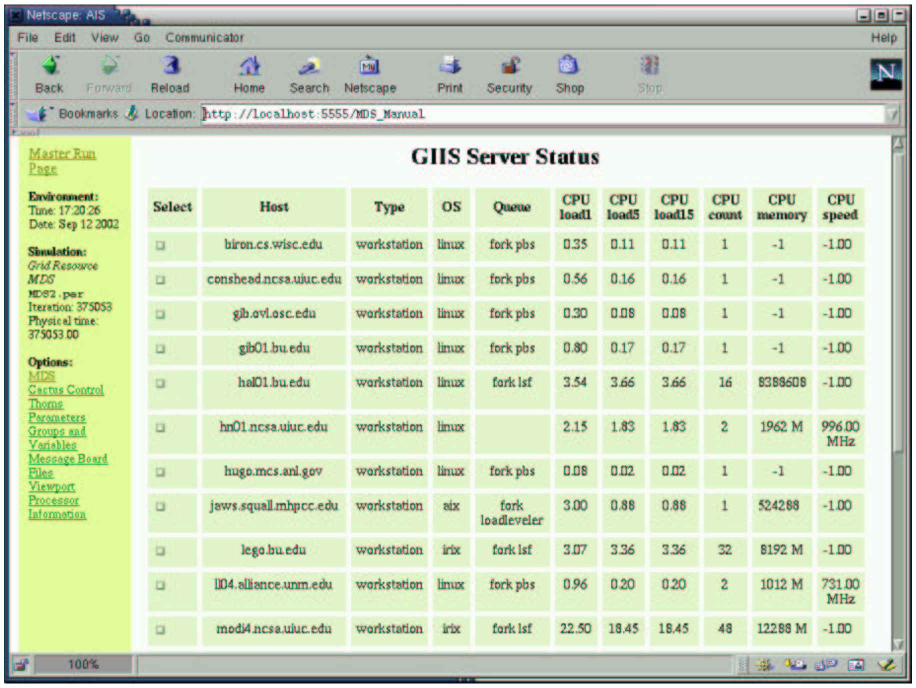
3. It can be used to supply additional data. In some cases it is important to have special information, for example about scratch file systems that have the capacity to store large checkpoint files.

7.5.3 Grid Resource MDS

Grid Resource MDS is an interface thorn that retrieves information from a Meta Directory Service [25]. The MDS is our primary mean to gather information about hosts in a Grid. The operation of the MDS server was discussed in Section 3.1.1. We have worked with MDS v1.1. The latest MDS v2.2 is part of Globus 2, which had just been released when we conducted our experiments.

The MDS thorn requests information from an arbitrary number of user specified MDS servers. The returned information is translated to a Grid Object and stored in the database of the thorn Grid Resource Base, which also provides the interface to access the data. Figure 7.7 shows the browser interface to the MDS data, now stored as Grid Objects.

The MDS is designed to store resource related information: MDS information on the deployed queue system allows the GMS to automatically chose the right batch submission syntax if Globus GRAM is not available. We see databases as an essential part in a Grid service infrastructure. Our experiences with the MDS indicate the enhancements that would make it even more effective. Our MDS queries were quite time intensive, taking up to a minute for a single server. Inconsistent data as


Netscape: AIS
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:5555/MDS_Manual
100%

Master Run Page

Environment:
Time: 17:20:26
Date: Sep 12 2002

Simulation:
Grid Resource MDS
MDS2.par
Iteration: 375053
Physical time: 375053.00

Options:
MDS
Cactus Control
Thorns
Parameters
Groups and Variables
Message Board
Files
Viewport
Processor Information

GIIS Server Status

Select	Host	Type	OS	Queue	CPU load1	CPU load5	CPU load15	CPU count	CPU memory	CPU speed
☐	biron.cs.wisc.edu	workstation	linux	fork pbs	0.35	0.11	0.11	1	-1	-1.00
☐	conshead.ncsa.uiuc.edu	workstation	linux	fork pbs	0.56	0.16	0.16	1	-1	-1.00
☐	gib.ovl.osc.edu	workstation	linux	fork pbs	0.30	0.08	0.08	1	-1	-1.00
☐	gib01.bu.edu	workstation	linux	fork pbs	0.80	0.17	0.17	1	-1	-1.00
☐	hal01.bu.edu	workstation	linux	fork lsf	3.54	3.66	3.66	16	8388608	-1.00
☐	hn01.ncsa.uiuc.edu	workstation	linux		2.15	1.83	1.83	2	1962 M	996.00 MHz
☐	hugo.mcs.anl.gov	workstation	linux	fork pbs	0.08	0.02	0.02	1	-1	-1.00
☐	jaws.squall.mhpcc.edu	workstation	aix	fork loadleveler	3.00	0.88	0.88	1	524288	-1.00
☐	lego.bu.edu	workstation	irix	fork lsf	3.07	3.36	3.36	32	8192 M	-1.00
☐	ll04.alliance.unm.edu	workstation	linux	fork pbs	0.96	0.20	0.20	2	1012 M	731.00 MHz
☐	modi4.ncsa.uiuc.edu	workstation	irix	fork lsf	22.50	18.45	18.45	48	12288 M	-1.00

Figure 7.7: The MDS interface of the Grid Resource services queries different MDS servers. The returned data for each resource is stored as a Grid Object. The browser interface shows some of the retrieved data. Sometimes MDS information is not consistent ("workstation") and incomplete. Manual completion resolves these problems.

shown in Figure 7.7 (machine type: "workstation") was another problem. Many clusters with MDS only reported data for the front node and not the full cluster. These problems can be contributed to the administrative difficulties of maintaining a distributed environment and are not technically insufficiencies of MDS.

MDS could become even more valuable, if it were possible to have more information in the database: For example, on the possible access methods (e.g ssh, sftp) for a machine, on the host's filesystem and its capacities. Large files must be stored on special filesystems, usually not the home directory. We currently supply this information manually. Using a ping client, we could easily deploy a file system monitor (as shown in sec. 8.3), we but we feel that MDS is the adequate place to store file system related information. The current implementation of MDS relies on GRIS servers to deposit information in the MDS database. It would be handy to allow any authorized application (like a monitor program) to deposit data (see our discussion on data contributions by autonomic applications in Section 3.7).

Our information requirements are somewhat special, they are usually not a problem for other scenarios. Some of these issues (like consistent MDS entries) are of administrative nature, rather than technical; others (like the database performance) have been addressed in most recent versions of MDS. We found MDS to be an extremely useful component for our service environment.

Chapter 8

High-Level Grid Migration Services

This chapter familiarizes the reader with the Grid Migration and Grid Spawn Service. We start with a detailed explanation of the migration service in Section 8.1 and continue with the spawn service in Section 8.2. We introduce a service monitor in Section 8.3 as a tool to supervise services and applications and restart them if necessary. We conclude this chapter with a discussion of open issues and future research fields for migration and spawning in Section 8.4. Migration and spawn services exhibit a similar program flow. Techniques discussed for migration also apply for spawning. The migration service environment provides the following capabilities:

- Migration and spawn servers are contacted by an autonomic client-application through a web service interface, requesting migration, spawn or other services.
- Clients can specify required and optional information as Grid Object arguments.
- The migration server uses redundant fundamental peer services to select resources, stage files and execute jobs. It is able to survive failing service instances.
- The migration server's design is able to handle failures during the migration, either by repeating an individual operation or by reiterating through the migration process with a new resource.
- The migration server offers a reliable startup verification. It uses the AIS to determine whether a migrated application has restarted successfully.
- The migration server can be used in a variety of ways, of which we demonstrate three in Chapter 9: migration, spawning and auto-recovery.

Our approach offers application-level migration to autonomic clients. While a kernel-level migration (as done with Condor) happens transparently to the application, we require a migration interface for the client, which requests the migration from a server. It is the responsibility of the client to provide checkpointing and basic communication capabilities. With the three test cases that we introduce in Chapter 9, we show that a traditional program can be easily upgraded to an "autonomic" application and perform migrations.

8.1 Grid Migration Service

The Grid Migration and Spawn Service is a high-level compound service that relies on fundamental services. The migration process is invoked through a RPC by an application, which has decided to migrate. The migration service selects a new resource, it copies the necessary files from the old machine to the new one and continues the simulation. First migration experiments were conducted with a prototype, called the Cactus Worm [60]. The experiments with that model and its testbed [9] have greatly influenced the design of this migration framework [61].

We start this section with the introduction of the client interface of the GMS (Section 8.1.1), followed by the web service interface (Section 8.1.2). The complex migration process on the server side is discussed in detail in Section 8.1.3 and 8.1.4. Section 8.1.5 is devoted to the startup and fault tolerant properties of the GMS and Section 8.1.6 discusses types of migration failures.

8.1.1 Migration Client Application

A GPS migration client operates self-contained and makes the ultimate migration decision, based on local or external information. While a server can signal a migration "suggestion" to a client, it remains with the client to follow such a proposal. A migration can e.g. be triggered when the resource consumption approaches the provided resource constraints of a queue. The expiration of granted queue time is such an example. We studied a migration condition based on the local load of a system in [4]. This approach has potential use for "cycle-stealing" in interactive environments. In batch queue environments the compute resource is usually not shared with other application and remains stable.

Note that this thesis is not focusing on the application's decision making process that leads to a migration. Such a process can be made arbitrarily complex. This thesis provides the service environment and supplies the client with an interface to request migration (and other services). We suggest and discuss basic migration and spawn policies for the client in Chapter 9.

The major steps of a migration for a client are shown as pseudo code below. Initially, the client composes a Grid Object that holds information like hosting machine, resource consumption, necessary restart files. The respective Grid Object containers are denoted (`MC`, `RC`, `FC`) and are attached to the Grid Object (`GO`). The client announce this data to an AIS (`ais_announce`). The client derives the local resource constraints that are dictated by the compute environment: e.g. the remaining compute time in a queue. The client stores this data in the resource container of a new Grid Object: `constrGO.RC`. Constraints and application information are updated in regular intervals (step 3.a and 3.b). If the current resource consumption drops below a critical threshold (step 3.c), a migration is triggered: In a first step the AIS is instructed to mark the application *inactive* (`ais_setstate`). The application writes its state to a checkpoint and requests the migration (`gms_migrate`). Note that the minimal migration algorithm consists of the construction of the Grid Object, the call to the application's checkpointing routine and the migration request. Everything else is optional.

```
Client:
1) Describe application through a Grid Object:  GO
   MC,RC,FC,SC: Machine, Resource, File and Service Container + Profiles

   GO.MC ← current host
   GO.RC ← current resource consumption
   GO.FC ← migration files (checkpoint, parameter, etc.)

2) Register with AIS:
   ais_announce(GO)

3)  Repeat

3.a)    Check resource constraints:
         constrGO.RC ← queue constraints

3.b)    Check resource consumption:
         GO.RC ← current resource consumption

3.c)    Compare current resource usage to the constraints:
        If  (GO.RC - constrGO.RC < tolerance)
           Inactivate Application:      ais_setstate(GO, inactive)
           Checkpoint Application State to File:  GO.FC
           Request Migration:           gms_migrate(GO)

   Until (Program finished)
```

As an enhancement the client may supervise the availability of a migration server by actively pinging the GMS or requesting frequent AIS updates. Figure 8.1 illustrates the request sequence from the

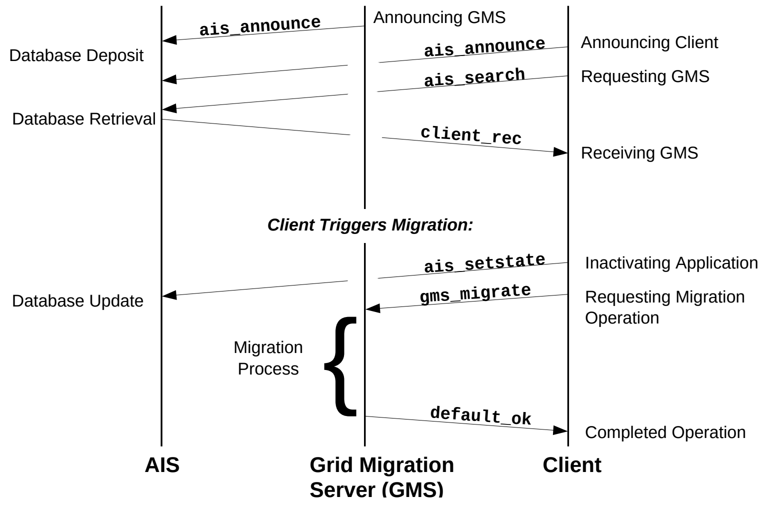


Figure 8.1: The Grid Migration Server as seen from a client application: The client contacts the AIS and requests the location of a GMS. Before relocation, the client inactivates itself at the AIS and requests a migration (`gms_migrate`). The client either waits for the reply of the error code through the default response method (`default_ok`) or terminates immediately.

client's point of view. Less important replies are not shown. The GMS announces its service to the AIS (`ais_announce`) and allows the client to retrieve the location of the migration server from the AIS by an `ais_search` request. Optionally, the client may announce to the AIS as well. In the event of an migration, the client deactivates itself at the AIS (`ais_setstate`) and sends a `gms_migrate` to the GMS. It may then wait for the arrival of the operation result as a default response methods (`default_ok`) or terminate immediately.

8.1.2 GMS Interface

The migration process appears as a rather simple event to the client application consisting of single service request, e.g. `gms_migrate`. Table 8.1 lists the most important GMS services and their arguments. The service calls are made by a client to the GMS. In most cases, the argument of a service call is a Grid Object. The server responds with a return value, indicating success or failure of the migration process. It is left to the application to receive and interpret the response.

GMS Operations: The GMS offers more functionality than just migration. We review the different GMS services as listed in Table 8.1:

- **Data Announcement**: Instead of requesting an immediate migration, the client announces relevant data without triggering a migration or any other operation: `gms_announce`. This is useful for applications that checkpoint in regular intervals but do not wish to migrate each time. An operation can be initiated anytime later with a request, specifying the unique ID (UID) under which the data is stored as a Grid Object.

- **Data Operation**: Announced data is untyped by default. To associate the data with a certain operation (migration, spawning, auto-recovery or storage), the data's operation type is set with `gms_settype` by the client.

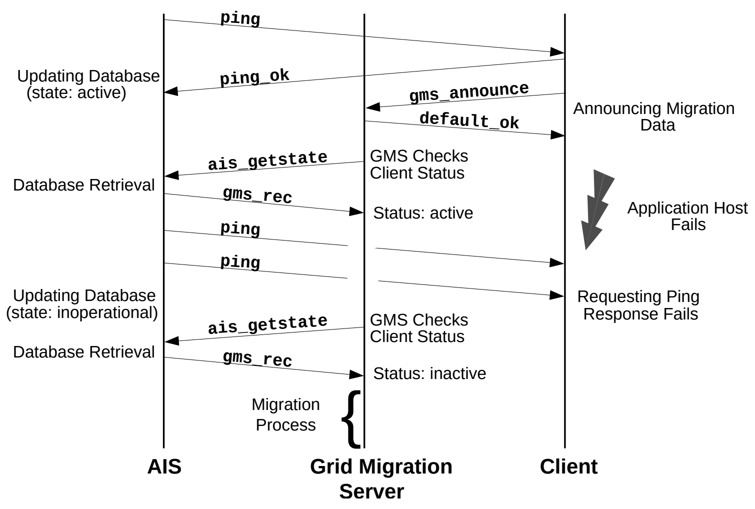


Figure 8.2: The GMS provides an *auto-recovery mode*, where it checks the AIS for the state of an application. If the AIS reports an *inoperational* state, the GMS proceeds to automatically recover the application on a new resource.

- **Migration**: The client requests a migration through a `gms_migrate` call to the GMS and provides a Grid Object with the data that is necessary to restart the application.

- **Automatic Recovery**: The Migration Server can be operated in a mode, where it requests the status information of the client application from an AIS. A client, which is declared *inoperational* by the AIS, can be *automatically* recovered from the program state of the last checkpoint. The application is then continued either on the same or on a new host. This mode of operation is illustrated in Figure 8.2. After two successive failures to receive the ping response, the AIS database changes the status of the entry to *inoperational*. The GMS requests the application's status from the AIS and starts the migration/recover process after finding the application inactive. In this mode, regular application checkpoints must be announced and the client activity must be traceable by the AIS.

- **Execution**: The GMS offers a simple interface (`gms_execute`) to execute an application on a remote machine, without startup verification.

- **Data Storage**: The GMS can be set up to move the announced data to secure storage through a `gms_store` call. This is helpful in the case that a site's queue policy only provides disk space to an application as long as it is executed in the batch system[1]. The application must ensure that the data is moved to a permanent file system *during the batch job*. This restriction can be handled by scripts *after* program execution – provided that enough queue time is left. The Grid Migration Service allows to deal with those files *during* runtime.

Migration Data: The client specifies those files in a file container, which are needed to restart the application on a remote resource (e.g. parameter and checkpoint files). If the client has the ability to profile its resource consumption, it can supply this data in a resource container as well. All information is collected in a Grid Object, which accompanies the service request. The following data is mandatory:

[1]For example, NCSA's queue policy.

Migration method	Arguments	Description
`gms_announce`	in: *Grid Object* out: *ID*	Input is the Grid Object defining the location of files on a client machine. The call does not associate a type (spawn, migrate) with the data.
`gms_settype`	in: *ID* out: *gms_type*	Sets the type of announced data: migration, spawn, storage data. Depending on the type, service operations vary.
`gms_migrate`	in: *Grid Object* out: *error code*	Initiates a migration. The Grid Object defines the files or holds the ID that identifies previously announced recover data.
`gms_autorec`	in: *ID* out: *error code*	Monitors a registered application and restarts the simulation of a failure is detected.
`gms_execute`	in: *Grid Object* out: *error code*	Copies data and executes a command on a remote machine, does *not* perform a success check with the AIS.
`gms_store`	in: *Grid File Object, Grid Service Object* out: *error code*	The specified Grid File Object is moved from the current machine to a new site, usually a storage facility, specified through a Grid Service Object.
`gms_rec`	in: *error code, Grid Object*	RPC routines to receive the feedback provided to the GMS.

Table 8.1: Grid Migration Server: Web Service Interface.

- **Execution Grammar:** specifies in which order an executable, parameter files, data files etc. are to be arranged to execute the startup command. In section 8.1.4 we give a description of a (simple) approach featured in the GMS.
- **Startup Files**: the client informs the GMS about its host machine with a *Machine Profile* and the location of the startup files with a *File Profile*. The specification of the access mode through a *Service Profile* is optional, since it cannot be expected that a client program knows about such details. The GMS will later complete the Grid Object with the appropriate access methods for the client host.
- **Executable:** the client specifies the name of the executable. The GMS provides the executable for the next host platform. Pre-staged executables or a repository of executables on a dedicated server are the supported methods in this version of the GMS.

The following information is optional and can be used to select an appropriate next machine:

- **Memory Requirements**: the application's memory requirements can be expressed in a *Resource Profile*.
- **Processor Requirements**: if a multiprocessor application requires a minimum number of processors it can express this in a *Resource Profile*.
- **Host Access Methods**: if the application knows about the local machine access, it can include this information in a *Service Profile*.

8.1.3 Migration Server

The migration server performs the migration on behalf of a client application. It receives the migration request and extracts the migration information from the transmitted Grid Object. The main steps of a migration are: **Resource Selection**, **Data Staging**, **Application Launch and Verification** and **Clean-Up**. These steps are shown as pseudo code below. The migratable object on the "old host" is stored in the Grid Object `oldGO` with the attached Machine, Resource, File and Service Container, denoted `MC`, `RC`, `FC`, `SC` respectively. "New" host information is stored in `newGO`. The GMS makes requests to fundamental peer services to select resources (`grb_match`), copy and delete files (`gfs_copy`, `gfs_delfile`), launch applications (`gss_submit`)) and retrieve information from the AIS (`ais_search`).

```
Server:
1) Receive migration request with Grid Object:  oldGO
   extract application information on "old" host:
     oldGO.MC ← old machine
     oldGO.FC ← checkpoint, parameter, etc.
     oldGO.RC ← resource requirements
     oldGO.SC ← various services

3) Determine new resource:
     newGO ← grb_match(GO.RC)

     newGO.FC  ←  oldGO.FC
     newGO.RC  ←  oldGO.RC
     newGO.UID ←  oldGO.UID

4)  If NOT ( oldGO.SC contains machine access)
       Determine access for old host:
       oldGO.SC ← ais_search(oldGO.[MC,SC])
```

```
5)  If NOT ( newGO.SC contains machine access)
      Determine access for new host:
       newGO.SC ← ais_search(newGO.[MC,SC])

6) Copy File to new host: gfs_copy(oldGO, newGO)

7) Start Application: gss_submit(newGO)

8) Check status: application state ← ais_getstate(newGO.UID)

9)  While (application state = inactive)

      Pause and Check again:
      application state ← ais_getstate(newGO.UID)

       If (application = inactive) after  N tries:
          Clean Target files: gfs_delfile(newGO)
          GOTO (Step 7)  OR (Step 6)  OR (Step 3)  OR quit loop

    End While

10)   If (application state = active)
       Clean Source files: gfs_delfile(oldGO)
        → migration OK

11)  If (application state = inactive)
        → migration FAILED
```

The left diagram in Figure 8.3 shows the migration process as a flow chart: in each migration phase external services are accessed in a non-blocking fashion. Each migration phase is processed according to the same state framework, which is shown in the diagram to the right: Any of the four migration stages is entered with the state `OPEN`. If the stage involves calls to external services, the migration is labeled `ACTIVE`, indicating that a response is expected from a third party. While states are `ACTIVE`, they are regularly examined for timeouts. If no service response is received within a given interval, the state is marked as `FAILED`. Depending on the state policy, a failed state can result in the following actions:

- The migration *phase* is repeated.
- The migration is rewound to an *earlier phase*.
- The migration event is declared a *failure*.

Note that the failure strategies illustrated here are implemented on top of the request failure policy that is provided by the underlying request handler (Chapter 6). The request handler makes best effort to execute an RPC or deliver a message.

Error Handling: Making the right choice, when to follow what failure policy is a field of research on its own: In general it is difficult to make an autonomic application conscious of *why* an operation fails. The cause of a failure may or may not go away with time: e.g. a copy operation might fail, because the network connection is taken down for 5 minutes or because the destination machine does not have enough disk space. A repetition of the copy operation may bring success in the first case, while it will not help in the second case. See the discussion on possible migration failures in Section 8.1.6.

The migration process can be made arbitrarily "intelligent" (and complex): a client announces its checkpoint files as a security measure against sudden failure and is automatically restarted when it

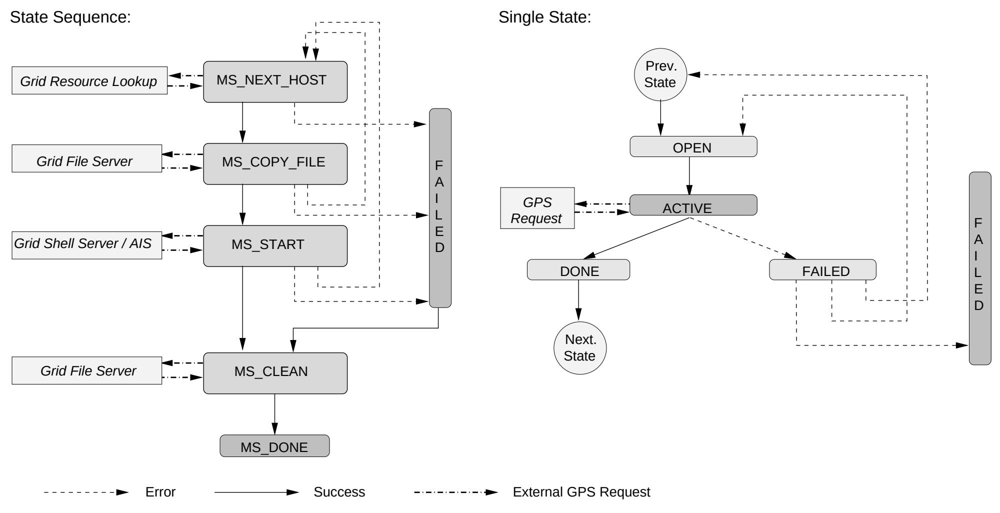


Figure 8.3: The migration state diagram. The right diagram shows the four migration phases and their external service access. The states of each step are show on the left diagram. The migration failure treatment is constructed on top of the RPC failure policies.

fails (see auto-recovery in Section 9.4.4). Other servers may constantly watch out for new resources and suggest more appropriate equipment to an application.

Resource Lookup: If a migration client has supplied a Resource Profile with the request, resource requirements are passed along to the resource lookup service. When an appropriate resource is returned to the migration service, the service checks if the machine and service data for the *target host* are complete. If this is not the case, the migration service must supply the missing pieces of information, e.g. on the machine access methods. The migration service also determines the access methods for the *source host*, if this has not been specified by the client. If the resource lookup service cannot provide an appropriate resource, the migration entry receives a time stamp and is paused, until a new lookup process is initiated. This query for computer capacities can be repeated several times. An exponential back-off provides an increasing timeout period to avoid successive failures.

Migration Data Staging: If a matching resource (e.g. a queue on a machine) is found, the migration data is transferred to the new host. Since the access methods for the new host are known, the migration server composes two Grid Objects: the first lists the migration files of the old host, the second gives the description of the new machine. These objects are passed along with a `gfs_copy` request to a Grid File Server. If all data has been staged to the new host, the migration server proceeds to restart the job.

Application Restart and Verification: The GMS composes the startup sequence and issues an execution command through a Grid Shell Server (GSS). The GSS executes the restart command on the remote host e.g. through a remote shell or by going through middleware installations like Globus GRAM, using the `globusrun` command. We currently do not use the Globus API but issue shell commands remotely.

The GMS is interpreting the return value of the restart request to check if the call succeeded technically. However, it would be shortsighted if the GMS identified a positive return code with a successful restart of the application for several reasons: If the applications starts off, but fails and shuts down later, the migration server would assume that the application is running, while it is actually dead. Furthermore, Globus execution methods provide ambiguous error messages, which are difficult to interpret, especially for service applications.

The GMS expects the restarted application to announce itself to the Application Information Server. The GMS queries the AIS until it receives an active-state response for the migration client. In this case the migration server assumes that the application has progressed far enough into its program flow and declares successful restart. The migration server can in theory ping the restarted application itself to determine if it continues. However, since the GMS has no information on which port the application is excepting request, it is usually a cleaner way to query the AIS to which the client announces.

Restart Timeout: The GMS queries the AIS for an active client state n times. The time interval is called the *restart timeout* and sets the time that is granted to an application to restart. This interval must be chosen with care. Potential errors can occur in the following circumstances:

- If the new resource is managed by a batch system, the application has to go through a queue wait period and may spend some time waiting to be restarted. A GMS which has set a very brief restart timeout would consider the application as failed, although it has not even started.

- If the query interval is chosen too long, the application might have already terminated (and hence inactivated itself) before the AIS is checked by the GMS.

- If the checkpoint files are of considerable size, the recovery progress may take longer than the GMS is willing to wait for the positive AIS feedback.

The theoretical wait time $T_{restart}$ is a sum of the client waiting time in the queue, recovery time, and announcing time:

$$T_{restart} = t_{queue} + t_{recover} + t_{announce}$$

To predict $T_{restart}$ we need to give the GMS some estimate of the possible wait time t_{queue} and startup time $t_{recover}$, while $t_{announce}$ is negligible ($t_{announce} \ll 1sec$ in our experiments). Very few schedulers are able to provide the queue wait time (e.g. Maui [68]). The recovery time in a multi-processor environment depends on the I/O performance of the disks and the network interconnects, if the data has to be distributed to all processors. Neither information is currently available in databases. A resource database, which tracks this kind of data, would be able to provide at least a simple prediction for recovery times on large scale supercomputers.

Migration Clean-Up and Data Storage: A migrating application can create a lot of files: checkpoints, parameter files, log files, etc. A clean-up process concludes the migration event. The clean-up process distinguishes between *target cleaning* and *source cleaning*: if a migration event fails in the startup phase, the target files on the next host must be erased, while the data sources are to be kept. They are needed for further restart attempts on other resources. If a migration event succeeds, the situation is reversed: source data files are deleted, while target data is used by the restarting application. The migration server cleans up its own data or data that has been announced with the migration. It is currently left to the client application to remove the files that it created itself. Note that we do not *move* restart files, but *copy* them to keep a working backup. A client can instruct the GMS to move files to central storage facility with `gms_store` requests.

8.1.4 Execution Grammar

Since the restart routine makes no assumption on how a program is launched on a host, the migration (or spawn) client provides this information as a *command template*. The startup routine parses the template and replaces all occurrences of `$<UID>$` with the file name of the profile that has the same UID. It also replaces `$<RUNCMD>$` with the architecture specific run command and number of processors (e.g. `mpirun`, `mpprun`, `poe`, etc.). This grammar deals with the startup cases that we have encountered, including sequences of commands.

The following example is a startup directive, which references the profile UIDs of the parameter file and two partial checkpoint files. The checkpoints are merged to a new checkpoint `temp.h5` with a `recombine` program before startup:

```
recombine $1242.3243$  $1242.3244$   temp.h5
$RUNCMD$ ./cactus_blackhole $1242.3242$ 2> temp.err > temp.log;
rm $1242.3243$  $1242.3244$ temp.h5
```

Note that the executables *recombiner* and *cactus_blackhole* can been addressed through their UIDs as well. Also note that a migration is not limited in the number of executables. It is possible to transfer any number of executables (two in this case) from a repository to a target host. $<*RUNCMD*>$ is translated into the host specific run command and the number of processors. The checkpoint files are erased afterward:

```
recombiner chkpt.it_0.file_0.h5 chkpt.it_0.file_1.h5 temp.h5
mpprun -n 4  ./cactus_blackhole BH_r3.2.par 2> temp.err > temp.log;
rm chkpt.it_0.file_0.h5 chkpt.it_0.file_1.h5 temp.h5
```

8.1.5 Startup and Fault Tolerance of the GMS

The migration server is only functional if its sub-services, which operate as independent peers, are ready to accept tasks. If the sub-services fail, the migration server cannot process migration requests properly. In such a case, pending migrations are not discarded but left in the queue for further processing. The GMS takes advantage of P2P service redundancy: it is not concerned where a fundamental services type is executed, as long as a service is available.

Figure 8.4 shows the temporal communication exchange for the startup phase and during runtime of a GMS, which only interacts with the AIS to monitor the availability of basic services like Grid File Services (GFS), Grid Shell Services (GSS) and Grid Resource Services (GRS). Before the GMS starts up, these fundamental services must be announced to the AIS (`ais_announce`). When the GMS starts up, it queries an AIS about the required sub-services (`ais_search`). The AIS returns the information (`gms_rec`) to the GMS, which adds the services to an internal database. Only if an active service instance for each these (copy, shell, resource) is reported by the AIS, the migration server is operational and announces its own services to the AIS (`ais_announce`).

Our Grid reliability study in Chapter 2 showed that any service implementation may shutdown unexpectedly due to software, hardware or network failure. This is also true for the fundamental services, on which a migration service relies. To recognize a change in the availability of the underlying services, the GMS sends an `ais_search` requests in regular intervals and updates its own database appropriately. An `ais_search` request is also triggered, if the request to one of the fundamental services fails. If the GMS fails to find the required services, it deactivates itself at the AIS (`ais_setstate`, not shown). It continues to query the AIS until all services are found, reactivates its AIS state and continues operation.

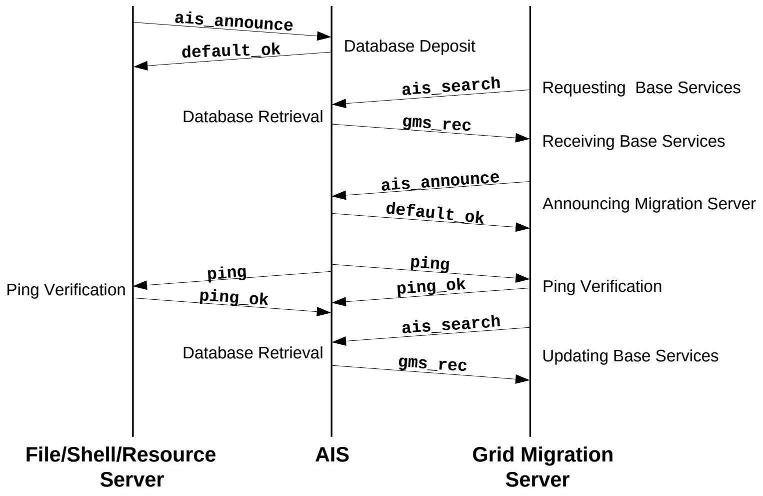


Figure 8.4: The Grid Migration Server relies on the availability of fundamental services. The GMS must ensure that these services are available to stay operational. It queries the AIS in regular interval to receive updated service information. The AIS tracks the announced services through ping requests.

8.1.6 Migration Failures

In this section we discuss the types of migration failures which we have come across. We also outline, whether these failures can be resolved by the migration server and how this can be done. The autonomic application gives up control to the migration environment, when a migration is initiated and migration failures must be handled on the server side.

Resource Failure: If a client specifies resource requirements that cannot be solved within a given time, the migration is aborted. The migration server makes several attempts to look up resources. Because queue and machine properties do not change within a day, a resource requirement which cannot be resolved in a first attempt, always failed in later lookups as well.

Binary Failure: If a binary for a new platform cannot be obtained, (e.g. in a binary repository), the migration to that host cannot continue. However, the server can attempt to identify a new resource, which satisfies the client's requirements and is not of the previous platform type (for which no binary existed). While this is not part of the current migration server, its implementation only requires additional resource profiles to blank out the failing platform in the Class-Ad selection process. This is an example of a failure which is "understood" by the service environment and it is able to take alternative actions.

Pseudo Failure: There is a chance for the GMS to react to pseudo-failures caused by bad timing of the ping response time and AIS search operation: If the ping response threshold is set too short, a brief failure in the network may lead to an *inoperational* state of the client in the AIS database, because ping replies do not get back in time. Since the AIS keeps pinging the client anyway, this state is quickly corrected with the first successful ping response. However, if the GMS sends an `ais_search` request to retrieve the client state from the AIS in this particular moment *and* does not verify the negative results with additional requests, it proclaims an application as failed, which is in reality still running. Since the GMS is instructed to retry the launch of the application or execute a recovery operation in the case of failure, a duplicate application instance is generated. To avoid such

Spawn method	Arguments	Description
`gms_spawn`	in: *Grid Object* out: *ID*	input is the Grid Object defining the spawn files, sets the type to "spawn" and initiates a spawning process. Can be used with `gms_announce`, `gms_settype`.

Table 8.2: Grid Spawn Server: Web Service Interface.

duplicates GMS makes sure that the old application is shutdown by trying to terminate it. Choosing the appropriate termination method is not trivial: the current implementation sends a termination request to the application. An explicit kill signal to the process ID or using the queue system's termination procedure would be the preferred way.

Algorithmic Failure: Automated recovery fails if the application aborts due to internal simulation errors rather than machine failure. The migration server still restarts the application, which dies again for the same reason. A possible work around is to stop automated recovery if a *progress metric* indicates no further simulation advance, for instance if the simulation is unable to continue further in time. This approach involves the comparison of a parameter whose name is provided by the client to the GMS (e.g. the iteration count). The client updates the parameter value in the AIS (`ais_setinfo`). The GMS retrieves the value of the progress metric (`ais_getinfo`) each time a restart is triggered and compares it to the previous value. If both values are the same, no advance in the evolution is assumed and the migration is abandoned.

8.2 Grid Spawn Service

The Grid Spawn Service is part of the GMS, since both services follow a similar order to move data and launch applications. Spawning describes the process of identifying routines in an simulation's program flow, which have no impact on the ongoing simulation and can therefore be executed independently of the main routine. The spawning of a subroutines pays off through the time (and money) that can be saved if expensive resources are only used for the core algorithms, while less demanding data analysis is performed on economic resources. An illustration of a spawning scenario is shown in Figure 2.2 on page 6 and a spawning example is studied in Chapter 9. The spawn service is requested by clients, who can provide a checkpoint and executable to restart the specific routine. The client's spawn process is similar to migration, shown in Section 8.1.1.

Spawn Alternatives: The alternative approach of storing the raw simulation data and performing a post-processing analysis is less satisfactory: while it speeds up the main computation, it can require a significant amount of disk space for the raw data. By spawning a routine, disk space is only needed during the transfer of the raw data to the external routines. The raw data can be deleted as soon as the routine is completed. An illustration of a spawning scenario is shown in Figure 2.2 on page 6 and a spawning example is studied in Chapter 9.

8.2.1 Grid Spawn Interface

Table 8.2 lists the spawn specific service calls, which operate like the migration service.

8.2.2 The Spawn Process

The client application contacts the Grid Spawn Service and provides the following information in a Grid File Object:

- **Execution Grammar**, which specifies in which order an executable, parameter files, data files etc. are to be specified.
- **Spawn Files**: the client informs the GMS about its host machine (*Machine Profile*) and the the spawn files (*File Profile*). The spawn files are usually smaller than in a migration event, because the total state of the simulation does not need to be saved. Only data important for the particular routine needs to be provided.
- **Executable:** the client specifies the name of the executable which continues the spawned routine. Similar to the migration, executables can be provided pre-staged or through a repository.
- **[Resource Requirements]**: optional – if the client has knowledge on the resource requirements by the spawned routine, it can express this in a *Resource Profile* to help finding the right resource.
- **[Host Access Methods]**: optional – if the application knows about the local machine access, it can include this information in a *Service Profile*.

Spawn data registration is possible via `gms_announce`, followed by a `gms_settype`(*spawn*) request. Similar to migration, `gms_spawn` requests trigger the spawn process immediately, while `gms_announce` stores the data without further actions.

Unique Identifiers: A major difference between a spawn event and a migration is the handling of the application UID. In the case of migration, the UID does not change. For spawning, new applications are created with each new task. Because the server must identify a spawn client with its parent (e.g. to transfer output data back to the parent), the spawn UID must be related to the parent UID. The spawned process inherits the identification number of the parent process as the base identifier and appends another UID. The spawned process registers and deregisters itself at the AIS under this new UID.

Spawn States: The spawn service traverses the same states as the migration service and features the same strategies for fault tolerance. Spawned applications are treated as independent applications: They can request auto-recovery or migration. As a difference to migration, the spawn server automatically transfers all data generated on the spawn host back to the parent host or another storage facility. The transfer relies on the assumption that all data, which is generated by the spawned application, is written into its current directory. The spawn server contacts a Grid Shell and File Service and requests the transfer of the spawn directory, either to the parent machine or to a user specified storage point.

8.3 Grid Service Monitor

The Grid Service Monitor is a distributed service application that maintains a constant number of services or applications on machines of a Grid. We implemented such a service by using the ping service in conjunction with a migration server as illustrated in Figure 8.5.

At start-up the service monitor instructs the GMS to launch an application on a number of hosts with `gms_execute` requests. The applications start up and register with the AIS. The service monitor

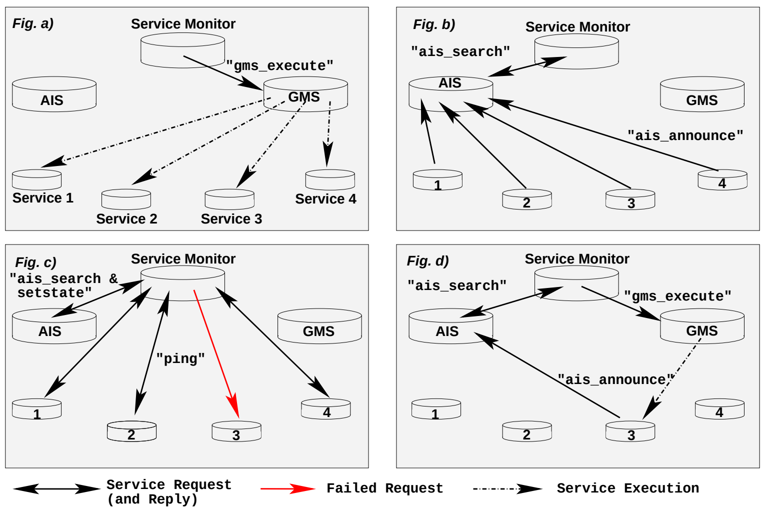


Figure 8.5: A service monitor watches the state of service clients and restarts them in case of failure.

checks the AIS for registered applications and monitors their state by issuing `ping` requests. If the client fails to respond to repeated ping requests, the monitor restarts the failed applications. The service monitor requires a traceable client application, either through ping requests or by keep-alive messages to the AIS.

Any GPS service (like GFS, GMS, GSS) can be loaded onto the application: We briefly elaborated on the idea of implementing a distributed Unique-ID server in Section 5.7. This service monitor is the appropriate tool to ensure its consistent availability. In Section 9.4.3 we demonstrate the automatic deployment of ping clients across machines of the EGrid. The service monitor is used to restart failing ping clients.

A “stacked” monitor concept supervises other service monitors to ensure constant operation of services. The stacked monitor approach is another step to prevent a single point of failures in distributed environments. The service monitor is not restricted to work with service applications: it is able to operated with legacy codes or proprietary applications like user simulations as well: in Section 9.4.4 we demonstrate how the service monitor is used to automatically recover a failing user application from backup checkpoints. In this case a single application is monitored.

8.4 Discussion and Future Research

In this section we discuss future extensions of the service environment that we presented in the previous two chapters. In our analysis of a Grid environment we rejected a monolithic approach in favor of a P2P service strategy to overcome failing service instances. To maintain a redundant set of services, we suggested a pool of information servers and introduced a service monitor, which supervises the various service instances and automatically restarts the ones that failed. The service monitor is designed to be used with services as well as user applications. Although the service environment has consolidated, it requires perfection and offers room for enhancements. We regard the improvement of the communication structure through OGSA, the extension of the service variety, and the increase of application “intelligence” as the important fields for future research.

8.4.1 Fault Tolerance for Autonomic Applications

The weakest participant in our migration service environment is not the service but the client application. While we circumvent single points of failures for services through redundant service deployment, we have only one instance of a client application. We currently rely on application checkpoints to restore a failed client. Since the continuous operation of a migrating client is the major goal, further means of safe-guarding the clients against failure must be found, perhaps through the coupling of kernel-level checkpoints with application level migration.

8.4.2 Authentication and Security

As we have stated initially, we have not concentrated on security issues in a distributed service environment. We feel that these issues are best addressed with upcoming service technologies like OGSA. Adding a security infrastructure to a distributed environment, which supports migration applications is not trivial. The following example illustrates the far reaching complexity of the security issue, which are yet unsolved:

- Applications must be authenticated before they request a service: if applications are started from a secure portal they can inherit its authentication. If they are launched interactively by the user, they must be authenticated "manually", which is a process where a user identifies an application as secure and trusting.
- Migrating applications come in two states: they alternate between an execution state and a checkpoint state. Nevertheless, they have to maintain an identity at all times, requiring the encryption of checkpoints files.
- Spawned applications must inherit their identity and authorization from a parent process. They must also maintain their identity during the checkpoint state.

8.4.3 Application Intelligence

Making applications and services more aware of their environment is important to operate in global Grids. The GPS services benchmark their own performance at every level. The Grid File Server e.g. stores the file transfer statistics to every host. Advanced Grid applications like the Cactus Code framework are able to profile memory consumption and processors performance. However, we are still looking for a proper evaluation of this data, e.g. to give preference to a slower machine with a fast network instead to a fast supercomputer, which has a low-grade network connection. With the upcoming *Network Profile*, we provide an extension to the Grid Object Description, which lets us express the results of different network benchmarking systems in unifying scheme. Network-based selection of resources is not crucial for intra-Grids, but it is important for global Grids with their shifting network loads. We regard assessment packages like Condor Class-Ads as essential for this task.

8.4.4 Using Advanced Grid Infrastructure

The modularity of the GPS implementation allows us to transparently add advanced Grid infrastructure like Grid-ftp with its reliable file transfer capabilities. Coupled with a prediction service to forecast the file transfer time, we aim at using intelligent file transfer in our environment. Advanced reservation of compute resources (e.g. provided through Maui) couples directly to file transfer time prediction. Achieving a nearly continues stream of resources requires a functioning advanced reservation system. The most widely used first-come, first-serve scheduling policy is too inferior and prone to abuse: after an excessive reservation of compute slots in various submission systems, only those slots are used, which are active by the time a previous slot expires - all others are discarded. Such an exploit system could easily be implemented with the migration service. We would like to offer this environment as a tool to evaluate different scheduling systems and charging algorithms with real-world applications.

8.4.5 Service Flow Control

It would be fascinating to investigate how the WSFL or its successor BPEL4WS (see Section 4.5.1) could be chosen to describe compound high-level services from the underlying fundamental GPS services. An important advance in the design of autonomic application would be the self-determined, dynamic definition of compound services, which would be created for a particular situation and destroyed afterwards.

Chapter 9

Grid Migration and Spawning Experiments

This chapter presents experiences and experiments that we conducted with the migration service environment, described in the previous chapters. We focus on two of the Grid case studies that we outlined in the introduction of this thesis: *Grid migration* and *application spawning*.

We start this chapter in Section 9.1 with a summary of the foundations that we have now at hand to experiment with migration and the spawning. We then acquaint the reader with the client applications: Section 9.2 describes a lightweight scalar wave evolution code and a numerical relativity simulation. Both are based on the Cactus Code and are developed at the Max Planck Institute for Gravitational Physics[1]. A genome analysis code, which is developed at the Technical University of Munich[2] is introduced in Section 9.3. In Section 9.4, we take a look at different migration, spawn and auto-recovery experiments with the numerical relativity client code. In Section 9.6 we describe migration experiences and visualization techniques for the genome analysis code.

9.1 Summarizing Migration and Spawn Infrastructure

We now have all the tools in place to realize the scenarios that have been our initial motivation: after analyzing the unreliable Grid infrastructure, we derived a service topology which possesses fault tolerant properties. We provided an information structure, which is able to present a compact view of objects on a Grid. We implemented a migration and spawn service environment as Grid Peer Services. Fundamental and high-level services are joined together to provide fault-tolerant task spawning and migration capabilities.

As mentioned we need to pose fundamental requirements to an application to make it eligible for migration: it must possess a hardware independence and it must be able to checkpoint and recover from a checkpoint. Our test applications fulfill these requirements. The migration experiments are carried out on a testbed of machines, which is assembled from various Computing Centers across Europe. All of these centers are part of the European Grid Initiative [30] and contribute to the Grid Lab project [10]. The different machines are listed in Appendix B.

9.2 Cactus Migration Clients

This section introduces Cactus based migration and spawn clients. In Section 9.2.1 we describe the modules (or “thorns”) that have to be compiled for a Cactus client. These thorns provide the application internal migration and spawn functionality to the application code (like file preparation) and allow to contact a migration or spawn service. In Section 9.2.2 we describe a simple test application, Section 9.2.3 introduces a numerical relativity simulation.

[1] `http://www.aei.mpg.de`
[2] `http://www.tu-muenchen.de`

9.2.1 Migration and Spawn Capabilities for a Cactus Client

As explained in Section 3.4, the Cactus Code enforces a modular “thorn” structure on the codes that are programmed in this framework. The modular approach has the advantage that all migration and spawning technology can be hidden in separate thorns, without touching any of the scientific simulation routines. In other words, the authors of the scalar wave code and the relativity simulation are not at all involved in the migration and spawning of their codes. Instead we are able to develop new capabilities and provide them to scientists for instant use.

Migration and Spawn Thorns

Below, we briefly list the modules, which contribute the migration and spawning interfaces to the client code and need to be added to the Cactus compilation.

- **WormBase**: This thorn provides the request handling routines, like extracting the envelope information, making the requested procedure calls and managing failing request and response messages (see Chapter 7 for details). The same module is also used by the Grid Migration and Spawn Services. It allows the client to send and receive RPC messages.

- **WormNG**: This thorn is in charge of communicating with the migration server. It queries the AIS for an active Grid Migration Server in regular intervals and publishes user information to a “personal AIS”. WormNG triggers the checkpoint procedure: a migration can be set off after a specified number of iteration, after a certain time interval or if a migration signal from an AIS is received. WormNG generates the restart parameter files, it expresses information on checkpoint and parameter files in a Grid Object and communicates with the migration server. WormNG in conjunction with Grid Ping supports auto-recovery.

- **Spawner**: The Spawner is a thorn, which discovers those routines in a Cactus executable that can be spawned. The initial spawner design was done by Allen [59]. The Spawner takes advantage of the scheduling system within Cactus. As explained in Section 3.4, all routines are scheduled and executed by the Cactus scheduler. The scheduler identifies the location of a routine in the program flow and the number and types of arguments that this routine receives. The Spawner is a *replacement* of the Cactus scheduler. For spawnable routines it initiates a checkpoint which contains exactly those variables, which would have been passed through in a function call. Routines, which cannot be spawned are executed in the traditional way. A parameter controls whether spawnable routines are actually spawned or executed internally. The Spawner creates a parameter file which instructs the executable to read in the checkpoint and execute the routine.

  The file information is passed to the WormNG module, which handles the communication with the spawn server. The spawn server transfers the data to a new host and restarts the application, which now only executes the spawned routine.

- **GridPing**: This thorn provides the Grid Ping Service capabilities as described in 7.2. It is optional and not required to request migration and spawning.

Other Required Thorns

The migration and spawn thorns rely on the following thorns, which are part of the Cactus Code framework. These thorns may depend on other thorns, which are not described here:

- **HTTP**: This thorn provides HTTP communication. It opens a port on the execution host and accepts HTTP Post and Send requests, e.g. from web browsers. Programmers can register functions with URLs, which are executed when the URL is requested. All GPS web pages shown in this thesis are such dynamically generated HTML code. WormBase uses the HTTP thorn to receive the incoming RPC request.
- **IOHDF5, IOHDF5Util**: The thorn IOHDF5 provides I/O capabilities in the HDF5 [53] format. It is used for checkpointing in the Cactus Code framework. IOHDF5Util's API is used to access the IOHDF5 data structure.
- **IOStreamedHDF5**: This thorn allows the streaming of checkpoint files from one application to another. It circumvents the writing, transferring and reading of checkpoint files. Checkpoint streaming requires an execution overlap between the sending, "old" simulation and the receiving, uninitialized code.

9.2.2 Scalar Wave Simulation

The scalar wave simulation is a lightweight application used to test the migration services. Before we target real-world applications, we use this program to analyze and examine the functionality of the migration service. The resource requirements of the test application are modest:

Test Simulation, small

Grid Size:	30^3
Flops per Grid Point:	12
Grid Functions:	7
Runtime:	∞
Total Memory:	1.4 MB
Checkpoint Size:	1.3 MB

Test Simulation, large

Grid Size:	60^3
Flops per Grid Point:	12
Grid Functions:	7
Runtime:	∞
Total Memory:	11.5 MB
Checkpoint Size:	10.4 MB

The scalar wave test simulation is implemented in the Cactus Code framework, which contributes checkpoint capabilities and platform independence. The thorns which enable migration for the scalar wave simulation are identical to those used for the numerical relativity simulations.

9.2.3 Numerical Relativity Simulation

Another client for our migration and spawning experiments is a numerical relativity simulation implemented with Cactus and developed at the Max Planck Institute for Gravitational Physics, Albert Einstein Institute (AEI). A research branch at the AEI focuses on the detailed description of relativistic phenomena through numerical simulation.

The detailed description of gravitating objects like binary black holes and gravitational waves [1, 2] is one of the most important problems facing relativity today: Binary black hole systems are the prime candidate for sources of strong gravitational waves. By the time the gravitational wave detectors like Geo600 [55] or LIGO [85] will come online, scientists who analyze the detector data for signs of gravitational waves, need filter patterns to know what to look for. The Einstein equations, which describe such relativistic scenarios, are a fully coupled elliptic-hyperbolic system of nonlinear partial differential equations. Finite difference methods on rectangular Cartesian meshes are used for discretization. The evolution equations are evolved in time with compute intensive schemes like leapfrog, McCormack and hyperbolic shock capturing.

9.3 Genome Analysis Migration Client

The third migration client is a philogenetic genome analysis program, called Grid Accelerated Maximum Likelihood (GAxMl) [63]. We start this section with an introduction of the algorithm and program (Section 9.3.1), followed by a description of the modifications made to allow it to communicate with the migration server (Section 9.3.2). Experimental results are discussed in Section 9.6. With GAxMl we show how minor modifications to a traditional, parallel program allow it to perform autonomic migrations in a Grid environment.

9.3.1 The Maximum Likelihood Algorithm

The purpose of philogenetic studies is to estimate the evolutionary genealogy of a group of species. It is used to reconstruct evolutionary ties between organisms and estimate the time of divergence since they last shared a common ancestor. The maximum likelihood approach is one of three major methods that are known to construct a philogenetic tree.

Like many problems in the field of genome analysis, the perfect philogeny problem is NP complete. Heuristics are introduced to reduce the search space in terms of potential tree topologies [33]. Still, philogenetic tree calculations remain computationally intensive: For instance out of the large amount of available data, like the 20,000 mitochondrial sequences in the ARB database [13], only a small fraction of data with sizes around 500 sequences have been compared.

Philogeny programs often feature a simple master-worker architecture, which makes the application easy to adapt to various resource situations. Their performance is primarily dictated by the number of available processing elements. Philogeny codes store their data in tree formats, which express the complex tree relationships and are comparatively small in size.

The master-worker paradigm requires data exchange with the worker at the beginning and ending of a search process. Unlike the solution process for partial differential equations, no synchronization of global variables or exchange of boundary values is required. The startup and checkpoint phases are brief and checkpoint sizes are in the order of megabyte.

9.3.2 Adding Migration Capabilities in GAxMl

The ancestor of GAxMl [63], called PAxMl [72] (“Parallel Accelerated Maximum Likelihood”) is a well tested philogeny program based on the master/worker paradigm and looks back on a history of successfully solved problems. Client migration capabilities were added in a collaboration by Stamatakis, TU Munich and Lanfermann, MPI Gravitational Physics. Inserting migration functionality was a challenge, since the application was not designed to be executed in an Grid environment. Program modifications had to be minimal to keep the researchers’ trust in the algorithm and program structure.

With GAxMl we want to demonstrate how migration capabilities can be added to user applications quickly without rewriting major portions of the code. We illustrate the minimal steps that we took to provide migration capabilities to PAxMl and called the resulting version GAxMl.

The original PAxMl code had no socket communication capabilities. While it is easy to add socket functions to permit *outgoing* communication, it is more intrusive to the execution flow to add socket polling capabilities. Because we wanted to show how only small changes to an existing and matured program are necessary for migration, we did not restructure the main execution flow of the program to accommodate socket polling procedures.

The GAxMl application starts with a *master* process, followed a by a *foreman* process, which communicates with a number of *worker* processes. The following additions to the master source code were made:

- **Registration**: GAxMl announces itself to the AIS and sends information about the machine that it currently executes on.
- **Runtime Information**: The application sends information, which reflects the current search state. This information is published on the personal AIS to be viewed by the scientist.
- **Migration**: If a migration is initiated, GAxMl writes the search tree to a checkpoint, composes the startup command and specifies the resource requirements. This information is sent to the GMS as part of the migration request.

The GAxMl application specifies the following information in Grid Objects and passes it along with the migration request.

- *Checkpoint Files*: The client declares files, which are required to restart the program. This is currently only two checkpoint files, for later versions of GAxMl a parameter will be added.
- *Executable:* The client informs the server about the executable that is used to restart the program. If the application is moved to a different machine architecture, the GMS retrieves an appropriate executable from a binary repository.
- *Startup Command*: Because the server has no knowledge on how the program wants to be started, GAxMl informs the server on its startup sequence.
- *Resource Requirements*: The client specifies its resource requirements. GAxMl requires a minimum of three processors to launch the master, foreman and worker processes. This information is expressed in the resource profile of a Grid Object.

Adding Communication: To permit the registration with the AIS and the communication with the Grid Migration Server, a number of routines were provided to perform the socket operations and allow the transmission of a request. The connection routines open a socket connection to a communication endpoint, which is provided by the AIS or GMS. The transfer routines take a request, serialize the XML code and embed the data in a HTTP header structure. This buffer is written to the socket and received by the server. The server deserializes the data and proceeds according to the requested method.

Checkpoint Files Structure: The checkpoint file does not describe the full state of the simulation. The simulation state can only be restored in conjunction with the original startup sequence. Therefore we have to transfer two files: the current checkpoint file and the original sequence file. For illustration, we show an excerpt of an initial sequence files with the names of the bacteria strands on the left and their RNA sequence on the right [13]. This data is analyzed through philogenetic matching and the results can be visualized as shown in Figure 9.14.

```
deinonema-  ATTTGCCCCA GGGATTCCCG CAAAAACCCC AGTAAGTTGG GGATGGCAGG GGAGGAA
ChlamydiaB  ATTTTCCCCA GAAATTCCCG AAAAAACCCC AATAAATTGG GGATGGCAGG GGAGGAA
flexistips  ATTTTCCCCA CAAAAAAAAG AAAAAACCCC AGTAAGTTGG GGATGGCAGG GGAGGAA
borrelia-b  ATTTGCCCCA GAAGTTAAAG CAAAAACCCC AATAAGTTGG GGATGGCAGG GGAGGAA
bacteroide  ATTTGCCCCA GAAATTCCCG CAAAAACCCC AGTAAATTGG GGATGGCAGG GGAGGAA
```

GAxMl Identification and Security: There is currently no room to store application related information with GAxMl. The GAxMl application as it stands now does not read a parameter file and the GAxMl checkpoint contains only tree related information. For now, we include information like application ID and generation count in the name of the checkpoint file. This is a temporary solution only, since it is not capable of transporting the information needed to provide basic security through a username/password scheme. The next version includes a parameter file, which contains application-ID, username, password, etc. This file may be encrypted and accompanies the migrating GAxMl code.

9.4 Cactus Migration Experiments

This section gives an overview of the different migration experiments conducted with the scalar wave and numerical relativity client. In Section 9.4.1 we show the result of a single-host reference migration, followed in Section 9.4.2 with results of a migration across selected machines of the European EGrid. We describe the implementation of a *service monitor* and demonstrate the results of a Grid ping measurement in Section 9.4.3. We conclude the migration experiments in Section 9.4.4 with the results of an auto-recovery experiment with a relativity simulation.

9.4.1 Migration Comparison Experiments

This section compares different transport and startup methods for the scalar wave test application. We want to establish reference benchmarks which are not effected by the characteristics of the Grid. Therefore we perform a “local” migration, in which the application is relocated on the *same* host between two different directories. The GMS still transfers files through scp etc. but we do not have to be concerned about the bandwidth fluctuation in an external network or different I/O performance of the systems. The reference migration was carried out 100 times on the host `origin.aei.mpg.de`, a 32 processor Origin 2000, running R10000 MIPS processors. The load of the system was negligible at the time the benchmarks were conducted. Timing results are averaged. The diagram in Figure 9.1 shows two reference migrations on the same host `origin.aei.mpg.de`. We show two applications with different mesh sizes: the migrating application to the left writes a checkpoint of 1.3 MByte, the application to the right has a checkpoint of 10.3 MByte. The GMS also transfers a complete executable of 5.3 MByte and a parameter file of 300 Byte. Timing results are shown for the *data transfer*, *execution*, *feedback* and *clean-up* phase.

Discussion: For Cactus based simulations, the most time-consuming phase is required for the data transfer. For the two checkpoint sizes shown, the total amount of data is dominated by the size of the executable. For larger checkpoint sizes, the size of the executable becomes less important. The code execution and cleaning is identical, it is governed by the speed at which the shell connection can be established. The feedback time is the second longest phase: an interactive restart procedure (not through a queue), which yields instant execution of the code. The feedback time depends on the pace at which the simulation is recovering from the checkpoint, followed by the registration with the AIS. We cannot directly identify the feedback time with the duration of the checkpoint recovery, even if we subtract registration overhead, for the following reason: The GMS checks the AIS with an exponential back-off strategy, starting with a two second poll interval, which is doubled on every second trial. Therefore, we see discrete instead of continuous feedback times: The first AIS poll returns an *inactive* status for the application, the second poll four seconds later returns *active*.

In Figure. 9.2 we compare the ssh based execution with a Globus/GRAM based submission for the application with the small checkpoint. There is virtually no overhead using a Globus based submission system over secure shell execution. We have seen that most of the time is used for the transfer of files.

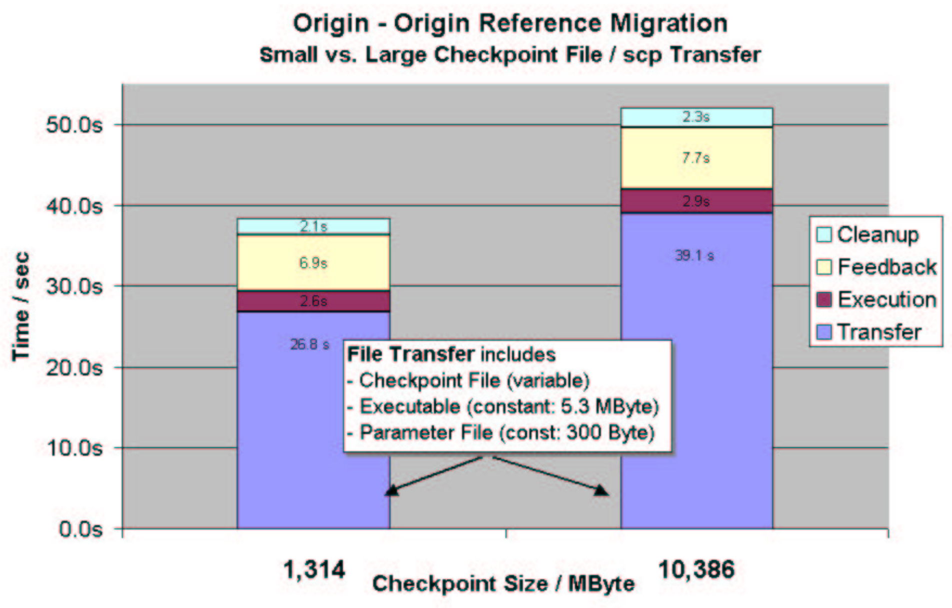


Figure 9.1: Origin-Origin reference migration for a small and large checkpoint file, showing the required time for the migration phases.

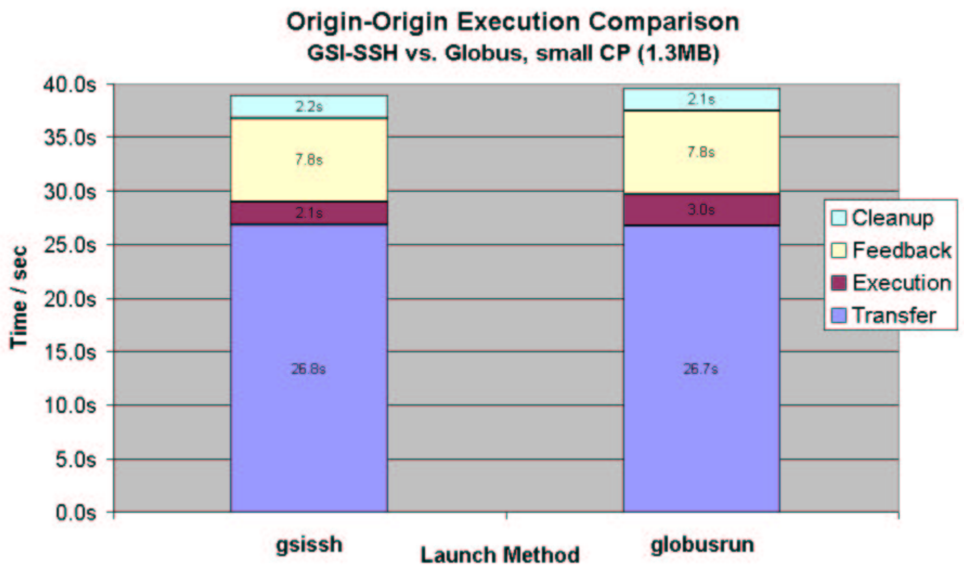


Figure 9.2: Comparison of execution methods: GSI-ssh based shell execution vs. Globus RSL script execution. Performance of the two methods is nearly identical.

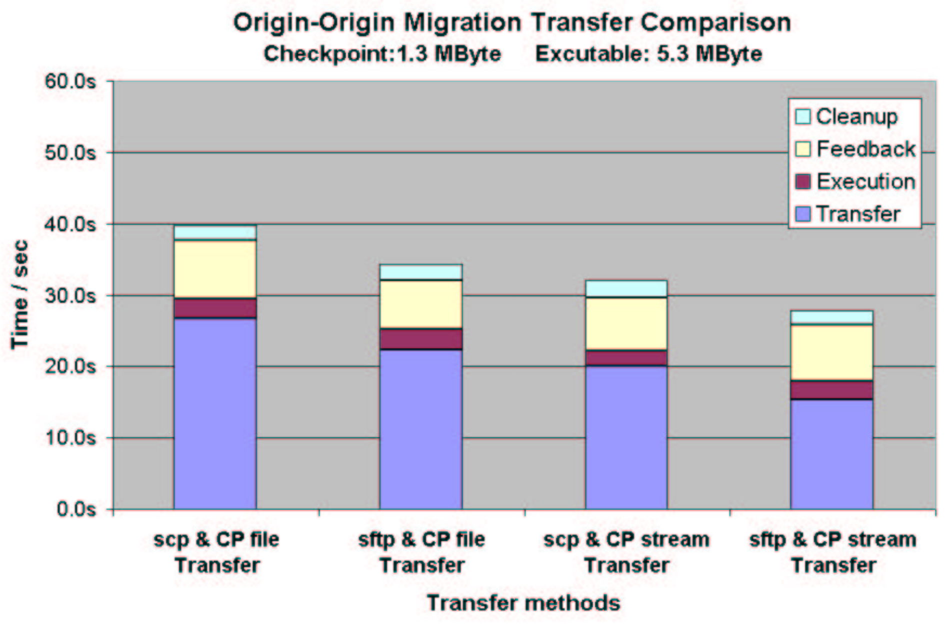


Figure 9.3: Comparison of transfer methods during the file copy phase for a small checkpoint file. Migration transfer is carried out by a combination of scp, sftp and checkpoint streaming. Streaming reduces the transfer time significantly.

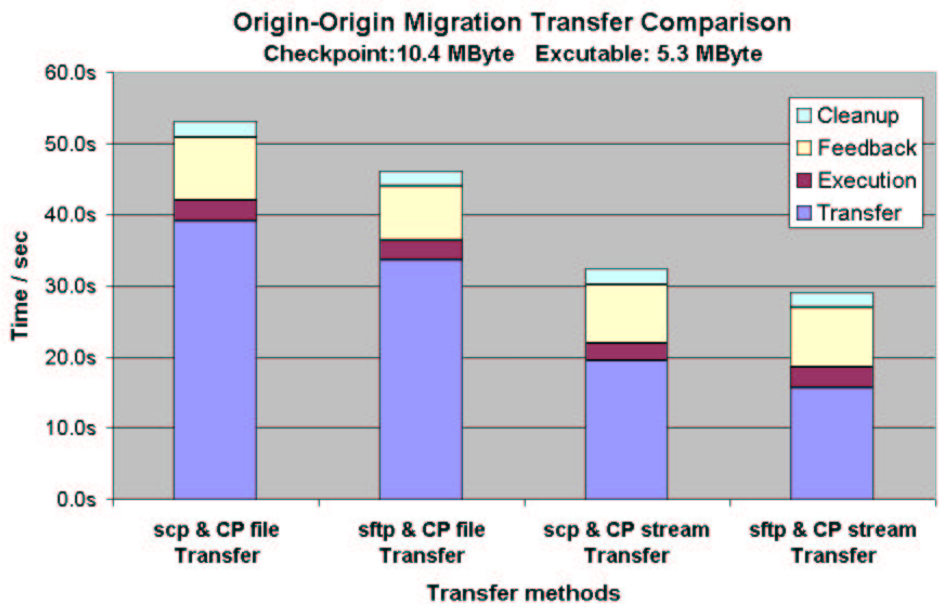


Figure 9.4: Comparison of transfer methods during the file copy phase for a large checkpoint file. Fastest migration can be achieved by combination of sftp for file transfer (executable and parameter) and checkpoint streaming.

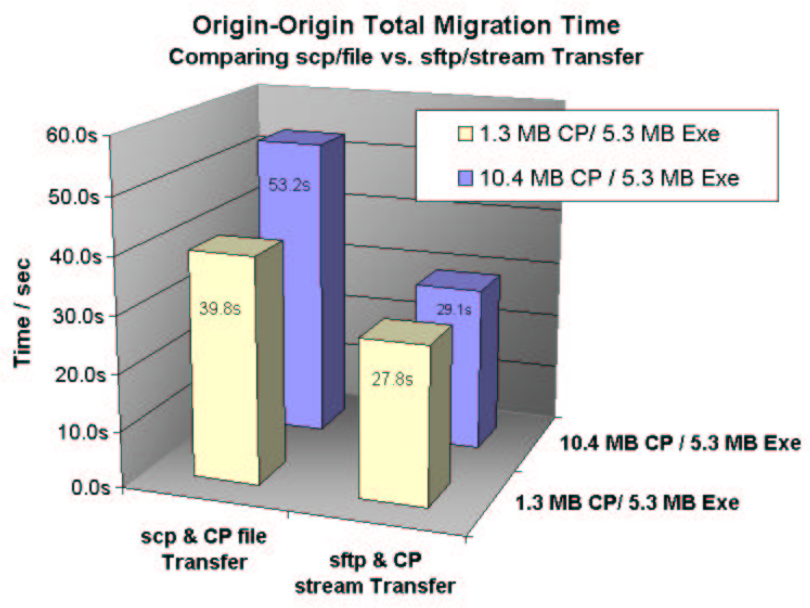


Figure 9.5: Comparison of the advantage of choosing sftp and streaming functionality over the standard scp file transfer method.

The overhead of checkpoint writing and reading is not captured here since it is part of the applications program flow. In Section 9.5 we analyze comparable I/O requirement for a spawning event.

In Figure 9.3 and 9.4, we compare several file transfer methods for the small and large checkpoint size, respectively. The first column shows a scp transfer for the executable and the checkpoint (abbreviated "CP") data. The second column shows the same data transfer accomplished through secure-ftp (sftp). sftp has the advantage that only the authentication is encrypted, while the actual data transfer is not, yielding a faster transfer rate. sftp is not necessarily available on all machines with ssh installation and sftp still requires a two-way copy operation.

To reduce transfer times even further, we experiment with the direct streaming of a checkpoint file from the expiring source simulation to the uninitialized target simulation. Checkpoint streaming requires an execution overlap between both application and open ports on both hosts. It is therefore not generally applicable to any migration, but still worth to look into. Column three and four of both diagrams show the file transfer for the executable and parameter files through scp and sftp methods, respectively, while the checkpoint (CP) is directly streamed. The relative speed of the migration increases the more the checkpoint dominates the total data size as shown in diagram 9.4. Note the long time spent in the feedback state, shown in column three of both graphs. This is the binning effect caused through the polling intervals. The migrating application has nevertheless started. In Figure 9.5, we summarize the speedup that was gained by using sftp and checkpoint streaming over a normal scp transfer. In Figure 9.5, we show the speedup that we gained by using sftp and checkpoint streaming over normal scp transfer.

9.4.2 EGrid Migration Experiments

In Figure 9.6 we show the results of a migration experiment carried out on machines on the EGrid. For each of the six hosts, we performed 20 migrations, resulting in 120 migrations over a time of

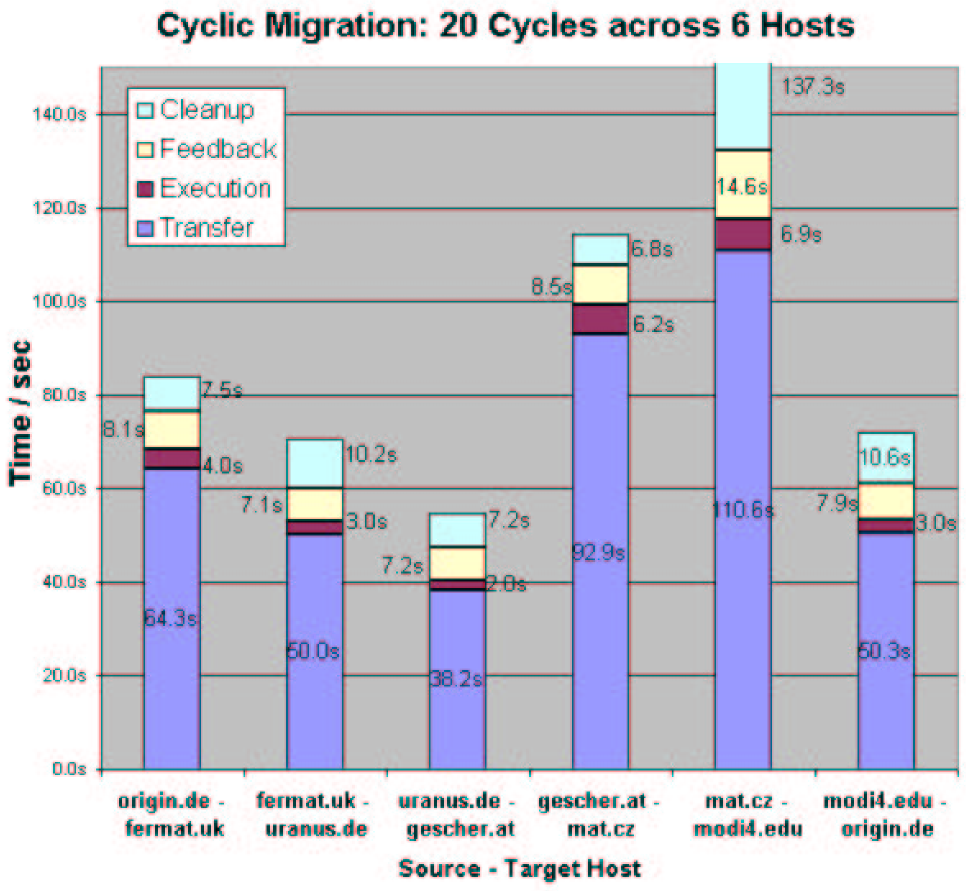


Figure 9.6: Cyclic migration across machines of the testbed. The x-axis denotes source and target host, abbreviated with hostname and top level domain.

approximately 20 hours.

Each column describes the migration between the two hosts, (hostname.top-level domain). The bars illustrate the time requirements for the migration phases, as reported by the Grid Migration Server. It is difficult to derive any sensible statements from this data, except that fluctuations are a fact: The variation of bandwidth was quite large. For example, the Clean-Up phase shown (column 5) invoked on `mat.ruk.cuni.cz` is based on a ssh access to the machine but still takes an excessive 137 seconds. This cannot be caused by a one time event, since we removed the worst and best data set before averaging the data. It might have been caused by an ill-configured ssh daemon with reverse name lookup problems. Transfer time includes file transfer duration from the source to the server and from the server to the target host. Since the request is made to a Grid File Server, the migration server only receives the result of the copy operation but cannot distinguish the two data transfers.

9.4.3 Ping Service Monitor

In Figure 9.7 we show a 75 minute excerpt of the ping statistics for ping clients that were deployed across machines in the EGrid. The ping reply times are effected by the network quality and load of the machine, because the ping clients are executed in user mode. The large fluctuations in reply time, especially seen in the bottom graph, can be contributed to shifting load on the machine.

The ping clients are supported through a service monitor (see 8.3), which automatically restarts failing applications. For `modi4.ncsa.uiuc.edu` note the regular interruption in the response time. This is caused by the host system terminating any interactive program after 15 minutes. On `uranus.cs.uni-potsdam.de` a similar interruption of unknown cause can be seen. The service monitor re-deploys a new ping client.

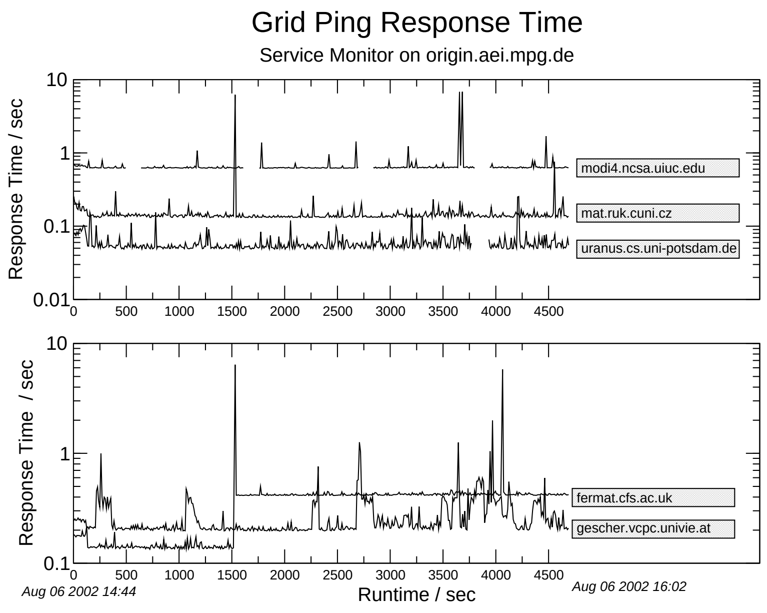


Figure 9.7: Ping reply numbers with automatic ping client restart. Extract of longterm ping monitor statistics.

9.4.4 Automatic Recovery

Client applications are the most vulnerable components in a distributed service environment. The service monitor offers an instrument to automatically restart client applications. Figure 9.8 shows an auto-recovery experiment conducted on `modi4.ncsa.uiuc.edu`. This host kills all interactive jobs after 15 minutes (as illustrated by the Grid Ping response times). We use this machine as our disruptive testbed. Figure 9.8 shows a sequence of 6 simulation phases. The simulation state as reported by the AIS is plotted at the bottom: it alternates between *INOP (inoperational)* and *ACTIVE*. The states are overlaid with the ping response time of the service monitor. In the initial phase, pinging does not start immediately: The ping monitor requests applications from the AIS in regular intervals. The absence of ping activity falls into this interval. The inset shows a blow-up around the time that the application is set *INOP*. Note that the ping responses are not received, but the application is still marked *ACTIVE*. The AIS requires multiple failing responses before an application is declared inoperational.

9.5 Cactus Spawn Experiments

In this section, we describe our spawn experiments and show under which conditions spawning either speeds up or deteriorates the performance of the main execution. As described in Section 8.2 “spawning” denotes the process of identifying routines in the program flow, which have no impact on the ongoing simulation and can be executed independently of the main routine.

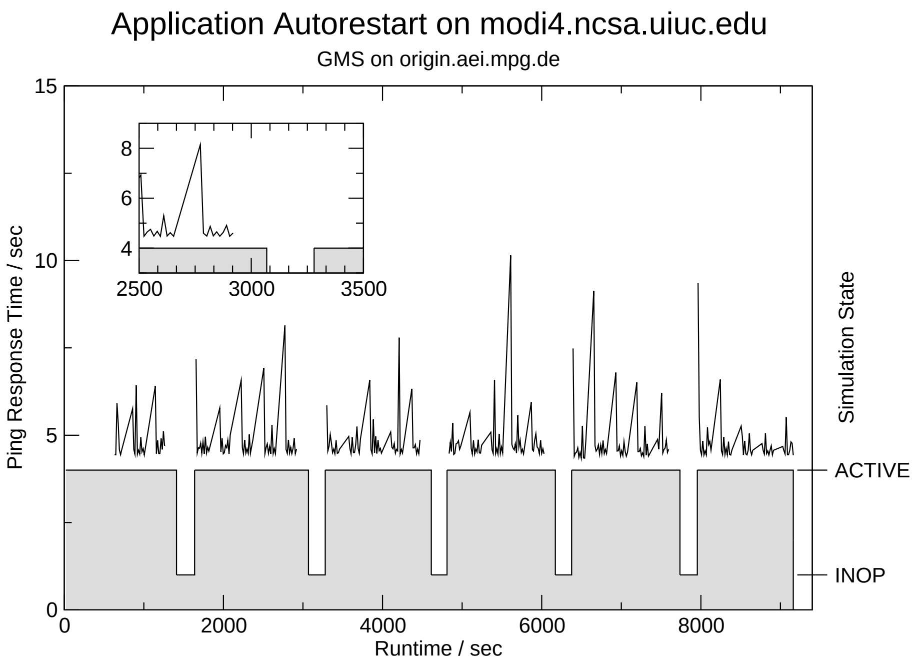


Figure 9.8: Automatic restart of an application on `modi4.ncsa.uiuc.edu`.

For the following experiment we use an apparent horizon finding (AHF) algorithm that operates on numerically evolved black hole data. We timed five successive AHF events. In Figure 9.9 we compare the spawned and internal execution of the AHF. The flow chart of the program is shown in the left inset of the figure. We compare the main program phases *Initialization*, *Evolution*, *Analysis*, *Communication*, *I/O* and *Total Time* for each of the three applications: the full simulation, which includes the *internal execution* of the analysis routine, the *spawn parent*, which spawns off the analysis routine and the *spawn client*, which executes *only* the analysis routine. The size of mesh was 10^3, resulting in checkpoint of 12.3 MByte per spawn event and with 84 mesh variables.

In this specific case we observe that spawning speeds up the main algorithm:

1. Spawning permits to reduce the total time consumed by the parent application. The parent's analysis bar includes the time spent to write the spawn checkpoint and is therefor not zero. The spawn parent may spend significantly shorter time on expensive compute resources.

2. Spawn clients finish fast: only the recovery operation and the analysis routine contribute to the total time. Other phases (like initialization or evolution) do not contribute.

3. Spawn clients are usually very lean: because they concentrate on a single routine, they can have a reduced memory consumption. Together with their fast execution time, they make a perfect application to fill idle machine cycles wherever possible.

4. Spawning does not necessarily reduce the *overall time* to complete the full simulation including the analysis process. This is expected due to the I/O overhead. Adding up the “total time” bars shows a extended execution for the full problem, compared to the internal execution. In this figure we did not include the duration for data transfer or waiting in the queue.

We describe a special case below, where spawning deteriorates the performance of the parent application.

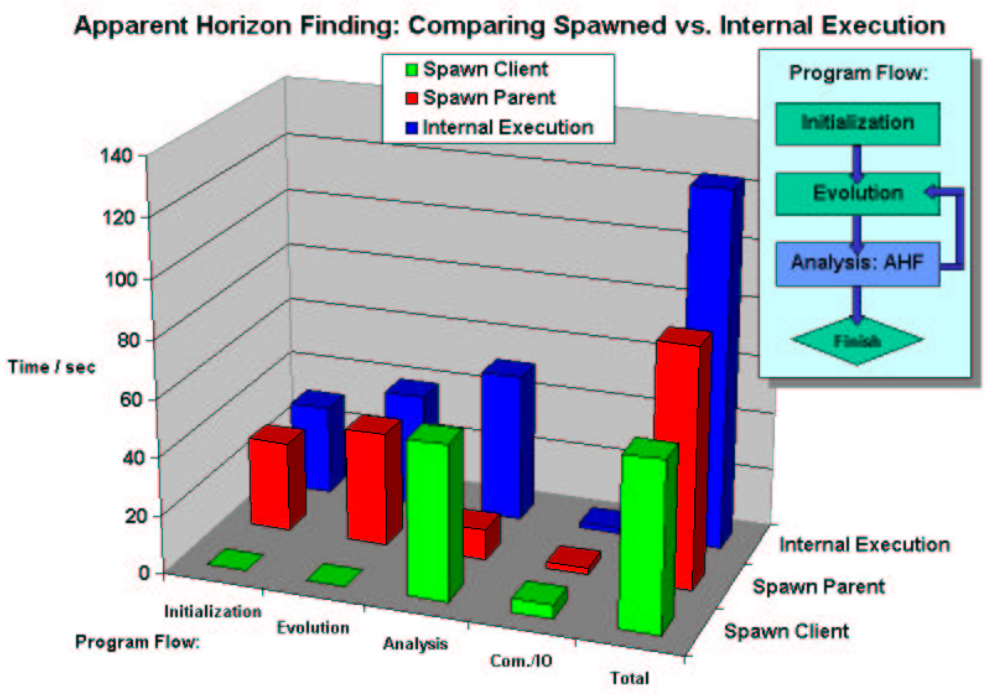


Figure 9.9: Comparing the spawned analysis routine to the internal execution: While the total time (spawn client plus parent) is larger than the total time spent for internal execution, the "parent time" on a main machine is reduced. This plot does not include the time spent for data transfer or queue wait time.

Critical Efficiency: Spawning introduces additional I/O for checkpoint writing and recovery and adds a network overhead for data transport, possibly even wait time in batch systems. These overheads a very dynamic: they depend on the network traffic, state of the queue, etc. The total overhead T_{spawn} can be estimated as follows:

$$T_{spawn} = t_{checkpoint} + t_{recovery} + t_{transfer} + t_{queue}$$

If the total overhead T_{spawn} is not compensated by a faster execution on an external machine, the overall time to calculate the problem will be higher than the time for an internal computation of the problem. This is expected and not surprising. However, spawning gives scientists an opportunity to minimize the simulation time spent on expensive high-level resources and perform analysis tasks on economic commodity resources.

Spawning vs. internal computation has a critical boundary: Application spawning severely harms the performance of the parent if the I/O process takes *longer* than the time spent for the internal calculation of the routine. We investigated such a critical case in Figure 9.10 and 9.11: Each of the two graphs shows the time (*y*-axis) spent for spawn-checkpoint I/O vs. the time for the internal calculation of the problem as the number of processors increases along the *x*-axis. We look at two counter-acting developments:

1. The spawn parent uses parallel I/O to generate the checkpoint file. The I/O time is slightly increasing with the number of processors. The I/O behavior depends on the client's I/O algorithm and the I/O load of the machine.

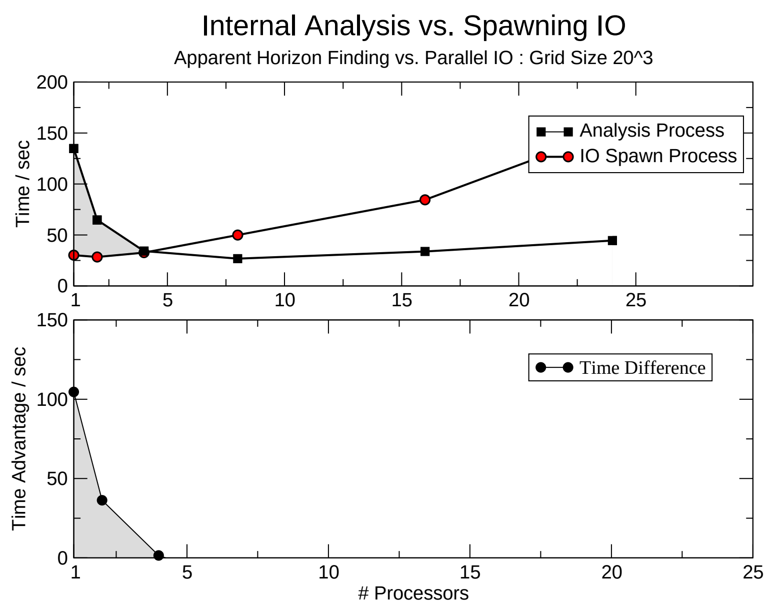


Figure 9.10: Time advantage of spawning off a routine compared to a local execution, grid size is 20^3. Here, a break even point is reached at four processors, where a local execution is faster than bringing spawn data to disk.

2. The analysis routine parallelizes fairly well up a certain number of processors: The compute time decreases as the number of processors becomes larger. For the small problem size of 20^3, the compute time has a minimum at 7 processors. Beyond that the computation takes longer as the number of processors increases, which is a well known behavior[3].

The bottom plot of both graphs shows the difference $\triangle T$, subtracting the internal execution time $t_{internal}$ from the duration of spawn-checkpoint I/O $t_{IOspawn}$. Note that we compare the I/O time of the spawn parent with the compute time of the internal execution. We do not look at the performance of the spawn client.

$$\triangle T = t_{IOspawn} - t_{internal}$$

At $\triangle T = 0$ we derive the critical number of processors P_{crit} where I/O takes as long as the internal calculation of the routine. Figure 9.10, with a mesh size of 20^3 shows the break even point of $P_{crit} \approx 4$ processors, while Figure 9.11 has $P_{crit} \approx 14$. The value of P_{crit} depends on two characteristics of the application: the parallelizability of the spawnable routine and of the I/O environment (I/O method and I/O load). P_{crit} is never reached if the spawnable routine requires more time on a single processor than the I/O process *and* if it possesses a lower degree of parallelization than the competing I/O process. Only case of $\triangle T > 0$ are practical for spawning. For $\triangle T < 0$ the preparation of the spawning (spawn I/O) takes longer then the internal execution.

It is a debatable how a spawning application determines the usefulness of spawning a routine: A first estimate of $t_{IOspawn}$ can be computed from the size of the checkpoint and the I/O characteristics of the device. Such data could be retrieved from an information server. In another approach the spawning application executes an experiment to determine $t_{internal}$ and $t_{IOspawn}$ and choses the method which is fastest. This approach is problematic when the behavior of the internal routine or the size of the spawn files changes in the course of the program flow. In such cases, the experiment must be repeated in regular intervals.

[3]Amdahl's law plus parallel overhead.

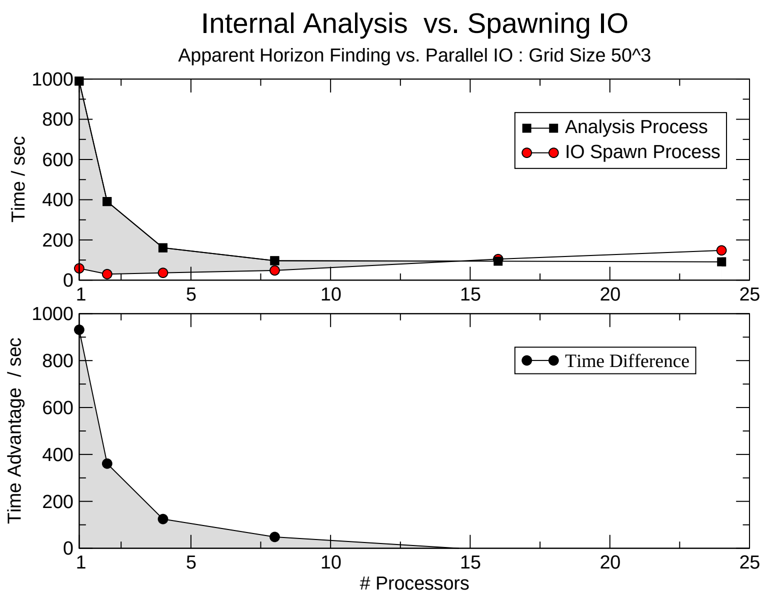


Figure 9.11: Time advantage for spawning off a routine compared to its local execution, mesh size is 50^3. A break even point is reached around 14 processors, where I/O takes the same time as internal execution.

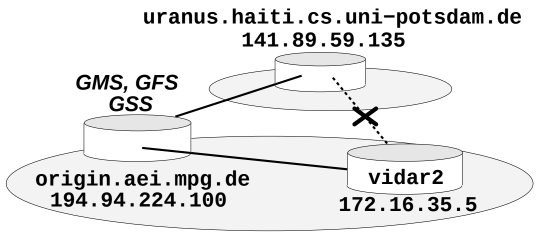


Figure 9.12: Migration of a GAxMl application between two hosts: While `uranus` is on a public network, `vidar2` is located on a private wireless LAN and only visible by `origin.aei.mpg.de`. Strategic placement of the Grid File and Grid Shell services (GFS, GSS) on common host enables migration between both hosts.

9.6 GAxMl Migration Experiments

This section describes the results of a migration experiments conducted with the genome matching code. In Section 9.6.1 we show the migration results, in the Section 9.6.2 we demonstrate visualization in a Grid environment.

9.6.1 GAxMl Migration

Figure 9.13 shows the results of a “ping-pong” GAxMl migration between two hosts. Note a slight asymmetry in the data transfer as the migration from `vidar2` to `uranus` shows faster data staging. The cause of this is not known. Unlike the migration on the EGrid, this relocation takes place between a publicly visible machine (`uranus.haiti.cs.uni-potsdam.de`) and a machine in a private wireless LAND (`vidar2`). There is no direct connection between both hosts. In this setup, the migration, file and shell servers are located on `origin.aei.mpg.de`, which has access to both resources. Figure 9.12 illustrates the service topology that allows the application to migrate between

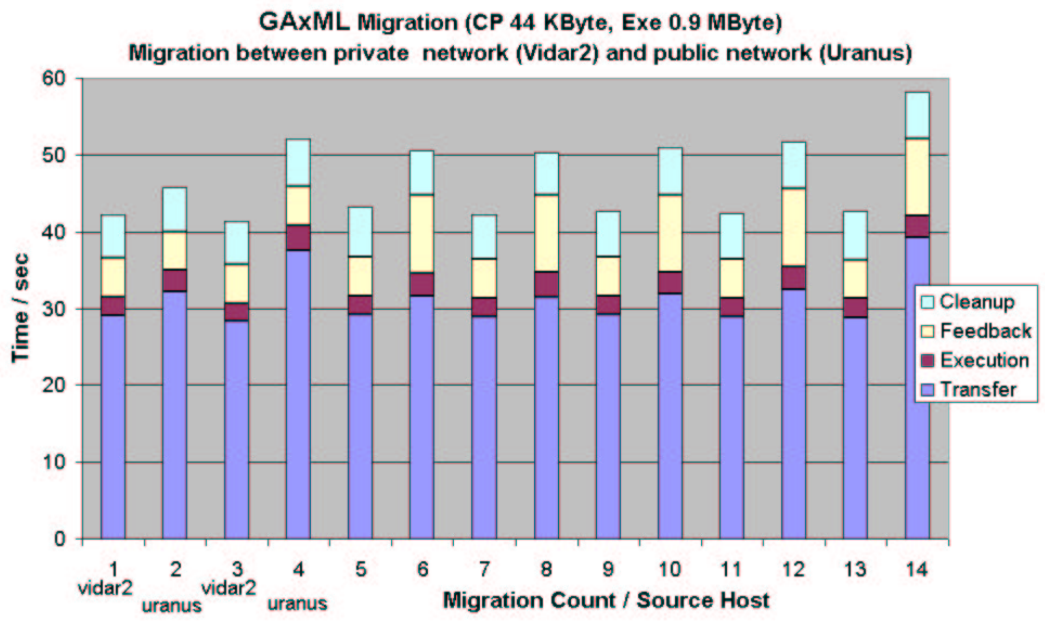


Figure 9.13: Alternating migration of a GAxMl application between two hosts. `uranus` is the head node of a cluster, `vidar2` a PC in a private wireless LAN.

hosts, which have no direct contact. In Section 9.7, we give discuss appropriate locations for the different services.

9.6.2 GAxMl Visualization on Grids

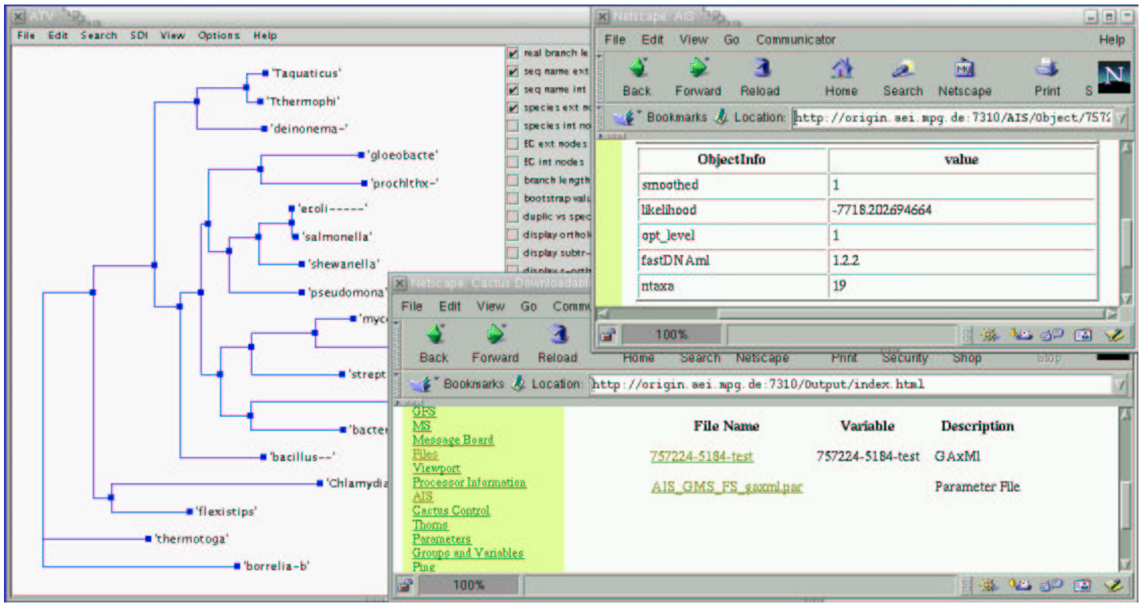
Grid migration can be regarded as an abstraction of the application execution from the application result; latter is of major interest to the scientist. In an automated migration environment it is not possible to tell where the application is currently running, or where it will execute next. These decisions depend on the shifting availability of resources. Nevertheless, the scientist has an interest in observing the simulation's progress and monitoring the results as they are generated.

For the GAxMl project, we worked out two solutions that permit scientists to track the progressing tree evaluation for a migrating application: The GAxMl application broadcasts information on the current genome matching state to the personal AIS, which advertises this information on its web page. In this mode, GAxMl uses `ais_info` requests and passes along the appropriate information as key/value pairs.

In an advanced approach, GAxMl makes use of the file advertisement features in Cactus [47]. Although GAxMl is *not* a Cactus applications, it can use Cactus visualization features indirectly through the pAIS: GAxMl sends the current tree to the pAIS in a `ais_info2file` request, along with a *MIME-Type* extension[4] In this case we request the extension *data/philo*.

When the pAIS receives the `ais_info2file` request, it writes the enclosed tree data to a file, associates the transmitted MIME-Type extension with it and publishes the URL on its web page. A web browser can be configured to startup special tree readers, when the user clicks on a link with this MIME-Type. Independently of where the genome code is currently executing, the scientist can

[4]The MIME-Type specifies how a web server is advertising a data file. Typical extensions are e.g. *application/postscript*, which instructs the browser to open a postscript viewing program.



ObjectInfo	value
smoothed	1
likelihood	-7718.202694664
opt_level	1
fastDNAml	1.2.2
ntaxa	19

File Name	Variable	Description
757224-5184-test	757224-5184-test	GAxMl
AIS_GMS_FS_gaxml.par		Parameter File

Figure 9.14: The screen shot shows a genome visualization program, which is launched automatically by clicking on the advertised file, shown in the lower left browser window. The upper right window shows client information on a personal AIS which reflects the current state of the matching process.

at all times inspect the ongoing process. In Figure 9.14 we show a screen shot of ATV[5], a Java tool to visualize philogenetic trees. The program is launched automatically by the browser. The lower right browser windows lists of the advertised tree and a parameter file. The upper windows shows information which helps researchers to monitor the progress of their application.

9.6.3 GAxMl Summary

With the GAxMl experiment we demonstrated the capability of the migration service environment to access resources in private networks. We elaborate on this concept in Section 9.7. We have also shown that an autonomic operation can be achieved by fairly “Grid unaware” applications. The more features and capabilities are available in a service environment, the fewer technology has to be loaded onto the clients. A web service interface is basically the only requirement for a client to participate in nomadic migration and use advanced visualization techniques.

9.7 Positioning of Grid Peer Services

Based on our experience we list the preferred location of the various servers:

- **AIS**: The Application Information Servers and its redundant instances must be positioned on machines, which are accessible from all participating resources.

- **pAIS**: The personal AIS’ must be positioned on a machines, which are reachable by an application. For convenience they can be located closely to the user to allow faster response. Multiple pAIS can serve a distributed research collaboration.

[5]`http://www.genetics.wustl.edu/eddy/atv/`

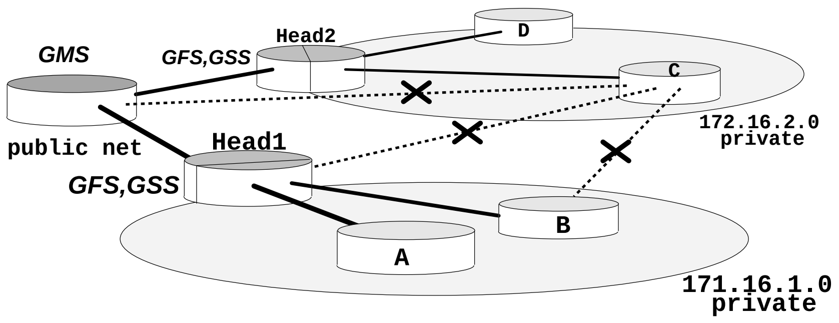


Figure 9.15: Grid File and Shell Services (GFS,GSS) can be positioned to allow access to resources on a private network. Such distribution of services does not require public IP numbers for resources

- **GRB**: The location of resource servers is rather arbitrary. They must hold contact to the migration server and be able to access third-party servers like MDS.
- **GMS**: the migration servers do not need to have direct contact to the migration resources, but must be reachable by a migrating application. The GMS delegates operations to the fundamental services, which makes it important that those have access to the sites.
- **GFS, GSS**: The file and shell servers must be located on machines, which have access to the resource that may host migrating applications. The GFS and GSS must stay in contact with the GMS.

Resources in Private Networks: To allow resource access in multi-domain environments with private networks, the GFS and GSS must be located at the interface between such networks. The two private networks `172.16.1.0` and `172.16.2.0` in Figure 9.15 are accessible from the public machines `Head1` and `Head2`, respectively. If these machines are chosen to host the shell and file server, a two stage copy operation to move a file from one private network to another. For instance, Condor-G connects its machine pools by using the Globus gatekeeper to access resources and the site's job scheduler. In our scenario, we go a step further since we access each resource directly. Note that if a Globus service is available on the head nodes, we can immediately copy to `Head1,2` and let Globus take it from there. A dynamic web service orchestration through WSFL or BPEL4WS as suggested in Section 8.4.5 would allow the proper coupling of copy services at runtime.

Chapter 10

Summary of Results and Future Research

In order to illustrate the contribution of this thesis, we review the past development style for Grid infrastructure and compare it with the possibilities that we now have at hand.

10.1 Grid Infrastructure Development

Grid Computing originated within the scientific and technical computing segment and software packages were often developed in isolated projects, which focused on special research aspects, like resource scheduling, file transfer, etc. The development style generated a hotchpotch of protocols and standards. Grid middleware required the installation of the same software on all hosts and was often not able to interact with third-party software of similar functionality.

As Grid research cooperations were formed Grid tools became less “standalone” and were put into a greater context. Still, the interoperability of today’s Grid infrastructure can be vastly improved. Applications (or “customers”) must become the driving force for Grid development.

The web service idea fits the world of Grid infrastructure ideally, as suggested by the Open Grid Services Architecture, which hides the proprietary implementation issues from the service functionality. However, many components of the current Grid infrastructure will remain non-webservice compliant for a long time. User applications are usually hand-coded and do not conform with the web service concept either. Nevertheless, legacy applications and user codes, must be incorporated in a global *Grid service environment* as well.

10.2 Contribution of this Thesis

The major results of this thesis can be summarized as follows:

- To describe the participants in complex scenarios, an generic data model for objects on a Grid is motivated. We propose the Grid Object Description Language in Chapter 5 as an information model to combine the different aspects of Grid entities.

- We implemented a toolkit to create and manage Grid Objects and use this tool in the implementation of our migration service environment.

- Global Grids are unreliable and require fault tolerant applications. We suggest the combination of web services with a Peer-To-Peer strategy, Chapter 4. With this union we gain a service and data redundancy and are able to overcome the failure of individual service instances. We call this fusion of two distinct service models “Grid Peer Services”.

- We presented a migration and spawn environment in Chapter 8 that is based on the Grid Peer Services idea. The environment is designed as a generic, modular and extensible service framework, based on the P2P topology. The service framework performs fundamental and complex services (e.g. copy and migration operations, respectively). It is hierarchically structured and makes use of existing Grid technology wherever possible.

- We presented the concept and implementation of an Application Information Server as an generic information registry to serve applications, services, file and resource related data. We introduced the personal AIS as a tool to view private simulation data independently of where a simulation code is executing.
- The presented migration environment enhances the throughput for long-term simulations.
- We tested the migration and spawn services with a Grid-aware simulation code based on the Cactus Code framework and a traditional, Grid-unaware genome analysis program. We showed that both are able to migrate autonomically.
- We analyzed under what circumstances spawning accelerates the execution of the core algorithm. We measured the critical processor number for a spawnable analysis routine in numerical relativity.
- We demonstrate that the migration service environment is able to access hidden resources in private networks.

10.3 Future Work

Throughout this thesis we have mentioned related work, pointed out extensions and outlined future projects. The concrete and long term research projects towards an environment that supports *autonomic computing* are:

- *Network Profiles:* Taking the changing network quality into account, we require an extension of the Grid Object data model, which describes the properties of networks.
- *Time Profile:* The current Grid Objects have no understanding of time or duration. We require such information to define e.g. the start and termination time of resources or services.
- *OGSA:* using OGSA conformal communication for the Grid Peer Services would allow us the offer a sophisticated security concept to migrating applications.

A new trend in distributed computing is emerging: autonomic computing. The self-determined operation of applications in service environments promises a new way to deal with the increasing complexity of compute resources. Making applications and service environment progressively more aware of their actions and failures and derive the proper consequences is essential for a truly self-governed and intelligent behavior.

Appendix A
GODsL Toolkit

The *Grid Object Description Language Toolkit (GODsL-Tk)* provides a set of routines which assist the programmer in manipulating the GODsL objects in the C programming language. In this chapter, we list the GODsL-Tk functions and give a brief example, on how GODsL-TK is used to send a migration request within a C program. The GODsL objects and the GODsL Toolkit are used to communicate the arguments for the various services introduced in Chapter 7.

A.1 Toolkit Functionality

The GODsL toolkit is written in in the C programming language. Other languages such as Perl, C++, are currently not implemented but the toolkit can be translated in a straightforward fashion. The GODsL-Tk provides APIs for object management and conversion of Grid Objects into an XML representation. The toolkit uses the xmlrpc-epi libraries[1], which conform to the XMLRPC specification [90]. GODsL-Tk provides routines to create, add, combine and delete profiles and containers. The GODsL-Tk specifically provides these functions:

A.2 Toolkit Programming Example

Listing A.1 gives an example on how the toolkit is used to handle migration files. This example is taken from a Grid migration client: The client collects information on all essential migration files in Grid Object `goMig`. The local routine `W_SetStructMachineProfile ("localhost")` stores hostname information in the machine profile `mp`. The routines `W_GetCheckpointInfo()`, `W_PrepareNextParfile()` and `W_GetExeInfo` gather information on the checkpoint, parameter file and executable, respectively. The machine profile `mp` and three file profiles are appended to the Grid Object `goMig`. This Grid Object is passed along as the argument of a `gms_migrate()` request to the migration server `migserver`.

A.3 Download

The migration and spawn service environment and GODsL-Tk is work in progress, please see the CVS section of the Cactus Code homepage `http://www.cactuscode.org` for download or contact `lanfer@aei.mpg.de`.

[1]xmlrpc-epi v5.0 by Dan Libby, Epinions.com `http://xmlrpc-epi.sourceforge.net`

`ToStructGridObject()`
`ToStruct{File,Service}{Profile,Cont.}()`
`ToStruct{Resource,Machine}{Profile,Cont.}()`
	Conversion of an profile,container or Grid Object in XML to its C structure representation. If no XML document is specified, the routine will create and initialize an empty structure.

`FreeGridObject()`
`Free{File,Service}{Profile,Cont.}()`
`Free{Resource,Machine}{Profile,Cont.}()`
	Releasing of allocated container, profile or Grid Object structures and free the allocated storage. The routine traverses into sub structures and frees all of the attached sub structures.

`CopyGridObject()`
`Copy{File,Service}{Profile,Cont.}()`
`Copy{Resource,Machine}{Profile,Cont.}()`
	The routines provide a copy the of the original object structure (profile, container or Grid Object) and return it to the programmer. The programmer is responsible for freeing this data.

`MC_AppMachineContainer(mc, mc)()`
`MC_AppMachineProfile(mc, mp)()`
`SC_AppServiceContainer(sc, sc)()`
`SC_AppServiceProfile(sc, sp)()`
`RC_AppResourceContainer(rc, rc)()`
`RC_AppResourceProfile(rc, rp)()`
`FC_AppFileContainer(fc, fc)()`
`FC_AppFileProfile(fc, fp)()`
	The first four routines appends take as the first argument a container structure (of type machine (`mc`), service (`sc`), resource (`rc`) or file (`fc`)) and append to this structure the content of the second container structure. The last four routines append a profile to a container of type machine (`mc`), service (`sc`), resource (`rc`) or file (`fc`). The routines are e.g. used to fuse multiple container structures into a single container.

`GO_AppMachineContainer(go, mc)()`
`GO_AppServiceContainer(go, sc)()`
`GO_AppResourceContainer(go, rc)()`
`GO_AppFileContainer(go, fc)()`
`GO_AppMachineProfile(go, mc)()`
`GO_AppServiceProfile(go, sc)()`
`GO_AppResourceProfile(go, rc)()`
`GO_AppFileProfile(go, fc)()`
	This set or routines appends a container or a profile structure to a Grid Object (`GO`).

`ToXMLGridObject()`
`ToXML{Resource,Machine}{Container,Profile}()`
`ToXML{File,Service}{Container,Profile}()`
	This set of routines converts the a profile, container or grid function structure into the corresponding XML structure. `ToXML` is the inverse to `ToStruct` routines.

Table A.1: GODsL-Tk overview, listing the different routines that are available to manage the Grid Object structures in C. The toolkit offers conversion routines to serialize C structures into their XML representation as well as deserialize XML to C structures.

```
GridObject          *goMig;
GOMachine_Profile *mp;
GOFile_Profile      *localcp, *localpar, *localexe;
char *reqID;

goMig = GO_ToStruct();
mp    = W_SetStructMachineProfile("localhost");

GO_AppMachineProfile(&goMig, mp);
goMig->key = strdup((char*)w_id);

localcp = W_GetCheckpointInfo();
GO_AppFileProfile(&goMig, localcp);

localpar= W_PrepareNextParfile(localcp);
GO_AppFileProfile(&goMig, localpar);

localexe = W_GetExeInfo();
GO_AppFileProfile(&goMig, localexe);

reqID = Wxml_NewRequest(migserver, goMig,
                        REPMODE_RESULT,
                        "gms_migrate", "default_ok");
```

Listing A.1: Migration Files: The Grid object `goMig` describes a set of migration files through a machine profile (`mp`) and three file profiles (`localcp`, `localpar`, `localexe`).

Appendix B

Grid Peer Service Testbed

The following machines were used to investigate the behavior of the Grid Peer Services and conduct the migration and spawn experiments. We give a brief description of the machines, which assembled this testbed.

Hostname	Institute	OS	Processor Type	Access Methods	Submission Methods
`origin.aei.mpg.de`	*MPI for Gravitational Physics*	IRIX	R10000	gsi, ssh	Globus
`modi4.ncsa.uiuc.edu`	*National Center for Supercomputer Applications*	IRIX	R12000	gsi, ssh	Globus, LSF
`uranus.haiti.cs.uni-potsdam.de`	*University of Potsdam, Computer Science*	Linux	Pentium 4	ssh	Globus, PBS
`gescher.vcpc.univie.ac.at`	*European Center for Parallel Computing, Vienna*	Linux	Pentium 3	gsi	PBS
`mat.ruk.cuni.cz`	*Charles University in Prague, Computer Science*	IRIX	R12000	ssh	Globus
`fermat.cfs.ac.uk`	*University of Manchester, Computation for Science (CfS)*	IRIX	R12000	ssh	–

Table B.1: Machines of the Testbed.

Index

Bibliography

[1] M. Alcubierre, G. Allen, B. Brügmann, G. Lanfermann, E. Seidel, W.-M. Suen, and M. Tobias. Gravitational collapse of gravitational waves in 3D numerical relativity. *Phys. Rev. D*, 61:041501 (R), 2000. gr-qc/9904013.

[2] M. Alcubierre, W. Benger, B. Brügmann, G. Lanfermann, L. Nerger, E. Seidel, and R. Takahashi. 3d grazing collision of two black holes. *Phys. Rev. Lett.*, 87:271103, 2001. gr-qc/0012079.

[3] W. Allcock, J. Bresnahan, I. Foster, L. Liming, and J. Link. GridFTP Update. Technical report, The Globus Project, January 2002. `http://www.globus.org/datagrid/gridftp.html`.

[4] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment. *Int. J. of High Performance Computing Applications*, 15(4), 2001. `http://www.cactuscode.org/Papers/IJSA_2001.pdf`.

[5] G. Allen, W. Benger, T. Dramlitsch, T. Goodale, H. Hege, G. Lanfermann, A. Merzky, T. Radke, and E. Seidel. Cactus Grid Computing: Review of current development. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *Europar 2001: Parallel Processing, Proceedings of 7th International Conference, Manchester*. Springer, 2001. `http://www.cactuscode.org/Papers/Europar01.ps.gz`.

[6] G. Allen, W. Benger, T. Goodale, H. Hege, G. Lanfermann, A. Merzky, T. Radke, and E. Seidel. The Cactus Code: A problem solving environment for the Grid. In *Proceedings of Ninth IEEE International Symposium on High Performance Distributed Computing, HPDC-9, Pittsburgh*, pages 253–260. IEEE Press, 2000. `http://www.cactuscode.org/Papers/HPDC9_2000.ps.gz`.

[7] G. Allen, W. Benger, T. Goodale, H. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. Cactus Tools for Grid Applications. *Cluster Computing*, 4:179–188, 2001. `http://www.cactuscode.org/Papers/CactusTools.ps.gz`.

[8] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficicient execution in heterogeneous distributed computing environments with Cactus and Globus. In *Proceedings of Supercomputing 2001, Denver*, 2001. http://www.cactuscode.org/Papers/GordonBell_2001.ps.gz.

[9] G. Allen, T. Dramlitsch, T. Goodale, G. Lanfermann, T. Radke, E. Seidel, T. Kielmann, K. Verstoep, Z. Balaton, P. Kacsuk, F. Szalai, J. Gehring, A. Keller, A. Streit, L. Matyska, M. Ruda, A. Krenek, H. Frese, H. Knipp, A. Merzky, A. Reinefeld, F. Schintke, B. Ludwiczak, J. Nabrzyski, J. Pukacki, H-P. Kersken, and M. Russell. Early experiences with the egrid testbed. In *IEEE International Symposium on Cluster Computing and the Grid*, 2001. Available at `http://www.cactuscode.org/Papers/CCGrid_2001.pdf.gz`.

[10] G. Allen and E. Seidel et.al. Gridlab: Enabling applications on the Grid. In *Proceedings of Grid 2002: 3rd International Workshop on Grid Computing*. Springer Verlag, November 2002. to be published.

[11] G. Allen, T. Goodale, G. Lanfermann, T. Radke, and E. Seidel. The Cactus Code: A problem solving environment for the Grid. In *Proceedings of First Egrid Meeting at ISTHMUS, Poznan, April 2000*, 2000. http://www.zib.de/visual/projects/TIKSL/Papers/EGrid2000-Cactus.ps.

[12] G. Allen, T. Goodale, G. Lanfermann, T. Radke, E. Seidel, W. Benger, C. Hege, A. Merzky, J. Massó, and J. Shalf. Solving einstein's equations on supercomputers. *IEEE Computer*, 32(12):52–59, 1999. `http://www.computer.org/computer/articles/einstein_1299_1.htm`.

[13] The ARB Project. TU Munich, 2002. `http://www.arb-home.de`.

[14] M. Bake. mpiJava: a Java interface to MPI. 1st UK Workshop on Java HKCN, 1998.

[15] A. Barak, A. Braverman, I. Gilderman, and O. Laaden. Performance of PVM with the MOSIX Preemptive Process Migration. In *Proceedings of the 7th Israeli Conference on Computer Systems and Software Engineering*, pages 38–45, Herzliya, June 1996.

[16] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metcomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[17] J. Basney, M. Livny, and T.Tannenbaum. High throughput computing with Condor. *HPCU News*, 1(2), June 1997.

[18] T. Bray, J. Paoli, C. Sperenberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0. W3C Recommendation, October 2000. `http://www.w3.org/TR/REC-xml`.

[19] E. Cerami. *Web Services Essentials*. O'Reilly Publishers, 1 edition, February 2002.

[20] E. Christensen, F. Curbera, G.Meredith, and S. Weerarawana. Web Service Description Language (WSDL). W3C Note 15, March 2001. `http://www.w3.org/TR/wsdl`.

[21] Common Information Model (CIM) Standards. The DMTF webpage: CIM Specification v2.7 and Standards, September 2002. `http://www.dmtf.org/standards/standard\_cim.php`.

[22] The Condor Classified Advertisement. The Condor Webpage. `http://www.cs.wisc.edu/condor/classad/`.

[23] Condor v6.3.1 manual. University of Wisconsin-Madison, 2001.

[24] The CORBA Specification. Object Management Group, November 2001. `http://www.corba.org`.

[25] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[26] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metasystems. *Lecture Notes on Computer Science*, 1998.

[27] The Data Grid Project. `http://www.eu-datagrid.org`.

[28] D.Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple Object Access Data Protocol (SOAP) 1.1. W3C Note, May 2000. `http://www.w3.org/TR/SOAP/`.

[29] Thomas Dramlitsch. *Distributed Computations in a Dynamic, Heterogenious Grid Environment*. PhD thesis, University of Potsdam, Potsdam, December 2002.

[30] EGrid Home Page `http://www.egrid.org`.

[31] A Grimshaw et.al. Legion: An operating system for wide-area-computing. *IEEE Computer*, 32(5), May 1999.

[32] H. Feider. Grid make. `http://www.cs.uni-potsdam.de/~schnor/potsdam/Research/Grid/grid\_make.html`.

[33] J. Felsenstein. Evolutionary trees from DNA sequences: A maximum likelihood approach. In *J. Mol. Evol*, volume 17, pages 368–376, 1981.

[34] R. Fielding, J. Gettys, J. Mogul, and H. FryStyk. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, Network Working Group, January 1997.

[35] T. Fokert, H.-P. Kersken, A. Schreiber, M. Striezel, and K.Wolf. The Distributed Engineering Framework TENT. In *VECPAR 2000*, pages 148–153, 2000.

[36] I. Foster and C. Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[37] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Service Architecture for distributed systems integration, June 2002. `http://www.globus.org/ogsa/`.

[38] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[39] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Intl J. Supercomputer Applications*, 15(3), 2001. `http://www.globus.org/research/papers/anatomy.pdf`.

[40] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Journal of Cluster Computing*, 5:237–246, 2002.

[41] C. Fricke. Characterizing networks through the Grid Object Description Language, 2002. Grosser Beleg, University of Potsdam.

[42] D. Gannon, K. Chiu, M. Govindaraju, and A. Slominski. An Analysis of the Open Grid Service Architecture. Technical report, commissioned by the UK e-Science Core Program, May 2002.

[43] Global Grid Forum Home Page `http://www.gridforum.org`, Applications Research Group home page: `http://www.zib.de/ggf/apps/`.

[44] The Gnutella Protocol Specification v0.4. `http://www.clip2.com/GnutellaProtocol04.pdf`.

[45] A. Gokhale, B. Kumar, and A. Sahuguet. Reinventing the wheel ? CORBA vs. Web Services. `http://www2002.org/CDROM/alternate/395/`.

[46] L. Gong. Project JXTA: A Technology Overview. Technical report, SUN Microsystems, April 2001. `http://www.jxta.org/white\_papers.html`.

[47] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing - VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin, 2002. Springer. to be published.

[48] Grid Adaptive Development Software. `http://www.isi.edu/grads/`.

[49] Globus Resource Allocation Manager. The Globus GRAM Webpage. `http://www.globus.org/gram`.

[50] Gridlab kick-off meeting, Poznan, February 2002.

[51] Gridlab: A grid application toolkit and testbed project. URL: `http://www.gridlab.org`.

[52] A. Grimshaw and W. Wulf. The Legion vision of a world-wide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[53] Hierarchical Data Format Version 5 (HDF5) Home Page `http://hdf.ncsa.uiuc.edu/HDF5`.

[54] R. Henderson and D. Tweten. Portable Batch System: External reference specification. Technical report, NASA Ames Research Center, 1996.

[55] J. Hough. Lisa: Laser interferometer space antenna for gravitational wave measurements, 1994. Prepared for Edoardo Amaldi Meeting on Gravitational Wave Experiments, Rome, Italy, 14-17 Jun 1994.

[56] Java Grande `http://www.javagrande.org`.

[57] Cactus Live Simulation Web Server. `http://tracker.aei.mpg.de:2000`.

[58] IBM. IBM Load Leveler: Users guide, September 1993.

[59] G. Lanfermann and G. Allen. Application Spawning: Resolving algorithmic dependencies. in preparation.

[60] G. Lanfermann, G. Allen, T. Radke, and E. Seidel. Nomadic migration: A new tool for dynamic grid computing. In *Proceedings of Tenth IEEE International Symposium on High Performance Distributed Computing, HPDC-10, San Francisco*, pages 435–436. IEEE Press, 2001. `http://www.cactuscode.org/Papers/HPDC10_2001_Worm.ps.gz`.

[61] G. Lanfermann, G. Allen, T. Radke, and E. Seidel. Nomadic migration: Fault tolerance in a disruptive grid environment. In *Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 280–281, 2002.

[62] G. Lanfermann, B. Schnor, and E.Seidel. Grid Object Description: Characterizing Grids. IFIP/IEEE Internation Symposium on Integrated Network Management, March 2003. to be published.

[63] G. Lanfermann and A. Stamatakis. Grid Accelerated Maximum Likelihood: Migrating philogentic analysis codes. in preparation, November 2002.

[64] F. Leymann. Web Service Flow Language (WSFL). Technical report, IBM, May 2001. `http://xml.coverpages.org/wsfl.html`.

[65] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis, using hardware counters. In *International Conference on Parallel and Distributed Computing Systems*, August 2001. `http://icl.cs.utk.edu/projects/papi`.

[66] M. Lorch and D. Kafura. Symphony - a Java based Composition and Manipulation Framework for Computational Grids. In *Proceedings Cluster Computing and the Grid (CCGrid 2002)*, pages 136–143, 2002.

[67] B.A. Mah. pchar: A tool for measuring internet path characteristics. `http://www.employees.org/˜bmah/Software/pchar/`.

[68] The Maui Scheduler. `http://supercluster.org/maui`.

[69] M. Migliardi, D. Kurzyniec, and V. Sunderam. Standards based heterogeneous metacomputing: The design of Harness ii. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS-HCW)*, Ft. Lauderdale, FL, April 2002. `http://www.mathcs.emory.edu/harness/`.

[70] M. Neary, B. Christansen, P. Capello, and K. Schauser. Javelin: Parallel computing on the internet. In *Future Generation Computer Systems*, volume 15, pages 656–674, 1999.

[71] G.R. Nudd, D.J. Kerbyson, E. Papaefstathiou, S.C. Perry, J.S. Harper, and D.V. Wilcox. PACE - A toolset for the performance prediction of parallel and distributed systems. *The International Journal of High Performance Computing Applications*, 14:228–251, 2000.

[72] G.J. Olsen, H. Matsuda, R. Hagstrom, and R. Overbeek. A tool for construction of phylogenetic trees of DNA sequences using maximum likelihood. In *Comput. Appl. Biosci*, volume 10, pages 41–48, 1994.

[73] S. Petri and H. Langendörfer. Load Balancing and Fault Tolerance in Workstation Clusters – Migrating Groups of Communicating Processes. *Operating Systems Review*, 29(4):25–36, October 1995.

[74] J. Postel. Internet Control Message Protocol. RFC 792, Network Working Group, September 1981.

[75] J. Postel and J. Reynolds. File Transfer Protocol (FTP). RFC 959, Network Working Group, October 1985.

[76] J.B. Postel. Simple Mail Transfer Protocol (SMTP). RFC 812, Network Working Group, August 1982.

[77] M. Ripeanu. Peer-to-Peer architecture case study: Gnutella network. In *Proceedings of 2001 IEEE International Conference on Peer-to-peer Computing*, 2001.

[78] L. Rose, Y. Zhang, and D. Reed. SvPablo: A Multi-language Performance Analysis System. *Computer Performance Evaluation (Tools)*, pages 352–355, 1998.

[79] M. Rose. The Blocks Extensible Exchange Protocol Core. RFC 3080, Network Working Group, March 2001.

[80] A. Sah. Symphony: A Java based Composition and Manipulation Framework for Distributed Legacy Resources. Master's thesis, Virginia Polytechnic Institute and State University, 1998.

[81] B. Schnor, S. Petri, R. Oleyniczak, and H. Langendörfer. Scheduling of Parallel Applications on Heterogeneous Workstation Clusters. In Koukou Yetongnon and Salim Hariri, editors, *Proceedings of the ISCA 9th International Conference on Parallel and Distributed Computing Systems*, volume 1, pages 330–337, Dijon, September 1996. ISCA, ISCA.

[82] E. Seidel, G. Allen, A. Merzky, and J. Nabrzyski. Gridlab — a grid application toolkit and testbed. *Future Generation Computer Systems*, 18:1143–1153, 2002.

[83] SUN Grid Engine Project. Sun Microsystems. `http://wwws.sun.com/software/gridware/`.

[84] L. Smarr and C. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[85] K. Thorne. Ligo, virgo, and the international network of laser-interferometer gravitational-wave detectors. In M. Sasaki, editor, *Proceedings of the Eight Nishinomiya-Yukawa Symposium on Relativistic Cosmology*, Japan, 1994. Universal Academy Press.

[86] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. The Grid Service Specification, February 2002. `http://www.globus.org/ogsa/`.

[87] S. Tuecke, D. Engert, I. Foster, V. Welch, M. Thompson, L. Pearlman, and C. Kesselman. Internet X.509 Public Key Infrastructure - Proxy Certificate Profile, July 2002. `http://www.gridforum.org/security/ggf5\_2002-07/draft-ggf-gsi-proxy-03.PDF`.

[88] Universal Description, Discovery and Integration (UDDI). `http://www.uddi.org`.

[89] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). Technical report, RFC 2251, Network Working Group, 1997.

[90] D. Winer. XML-RPC Specification, June 1999. `http://www.xmlrpc.com/spec`.

[91] R. Wolski. Dynamically forecasting network performance using the Network Weather Service. In *Cluster Computing*, pages 119–132, 1998.

[92] S. Zhou. LSF: Load sharing in large-scale heterogenous distributed systems. Workshop on Cluster Computing, 1992.