

---

# AstroGrid-D

Deliverable D3.1

---



## AstroGrid-D Distributed File Management Requirements Specification and Architectural Design<sup>1</sup>

Deliverable	WG-3 D.3.1
Authors	Working Group 3
Editors	Mikael Högvist, Thomas Röblitz, Thomas Radke, Angelika Reiser
Date	September 18, 2006
Document Version	1.0.0
Current Version	1.0.0
Previous Versions	Initial version
0.1.0, 0.1.1, 0.1.2, 0.1.3, 0.2.0, 0.2.1	0.1.0

### A: Status of this Document

Working draft.

### B: Reference to project plan

Deliverable D3.1: *AstroGrid-D Distributed File Management*.

---

<sup>1</sup>This work is part of the AstroGrid-D project and D-Grid. The project is funded by the German Federal Ministry of Education and Research (BMBF).

**C: Abstract**

The distributed file management in AstroGrid-D is developed to support the grid user when performing common tasks within the grid environment. For example, automated staging of input and output data, sharing of data within collaborations and monitoring job output. This document contains an analysis of the user requirements and a description of the architecture for the distributed file management.

**D: Change History**

<b>Version</b>	<b>Date</b>	<b>Name</b>	<b>Brief summary</b>
0.1.0	15.05.2006	Mikael Högqvist, Thomas Röblitz	Initial version
0.1.1	05.06.2006	Thomas Radke	fleshed out remote partial file access and interactive log-file access in architecture section
0.1.2	05.06.2006	Thomas Radke	use case descriptions for UC1..UC6
0.1.3	05.06.2006	Mikael Högqvist, Thomas Röblitz, Thomas Radke	Additions to the architecture section, interfaces, use cases, related work, ...
0.2.0	24.08.2006	Mikael Högqvist, Thomas Röblitz, Thomas Radke, Angelika Raiser	Changes according to feedback, re-iteration of the architecture section
0.2.1	25.08.2006	Mikael Högqvist, Thomas Röblitz, Thomas Radke, Angelika Raiser	Feedback changes, bibliography
1.0.0	18.09.2006	Mikael Högqvist	Changes according to feedback from the project

# Contents

Abstract . . . . .	2
Change History . . . . .	3
<b>1 Introduction</b>	<b>5</b>
1.1 Scope . . . . .	5
1.2 AstroGrid-D Architecture Overview . . . . .	5
1.3 Document Structure . . . . .	7
<b>2 Use Cases</b>	<b>8</b>
2.1 Simulations . . . . .	8
2.2 Analysis of data sets . . . . .	12
2.3 The Planck Process Coordinator (UC13) . . . . .	16
2.4 Integration of Robotic Telescopes . . . . .	16
<b>3 Requirements</b>	<b>18</b>
3.1 User Requirements . . . . .	18
3.2 Grid and Generic Requirements . . . . .	19
<b>4 Architecture of the Distributed File Management</b>	<b>20</b>
4.1 Storage layer overview . . . . .	21
4.2 Data objects and operations . . . . .	23
4.3 Storage layer nodes . . . . .	25
4.4 Overlay network design . . . . .	27
4.5 Grid nodes . . . . .	28
4.6 Security model . . . . .	29
4.7 Metadata . . . . .	29
<b>5 Service Interface</b>	<b>32</b>
5.1 Interface description . . . . .	32
5.2 Data types . . . . .	32
5.3 Exceptions . . . . .	32
<b>6 Scenarios</b>	<b>36</b>
6.1 File access using an OID . . . . .	36
6.2 Accessing a replicated data object . . . . .	37
6.3 Upload a data object . . . . .	37
6.4 Partial HDF5 file access . . . . .	38
References . . . . .	39

# 1 Introduction

AstroGrid-D is a grid initiative with the main goal of creating a grid-computing platform supporting applications from the astrophysics community. These applications originate from scientists in different German research institutes. Each member institute provides resources which will be part of the final grid environment. The applications and resources are highly heterogeneous with simulation code running on single workstations to parallel high-performance computers. Furthermore, AstroGrid-D include raw data producing equipment such as robotic telescopes and applications analyzing data which require high-throughput environments.

Files and data sets within the AstroGrid-D are geographically distributed over a wide range of systems with different access methods and network capabilities. Moreover, data sets are placed within different administrative domains where the file management must acknowledge local policies for accessing data sets. This introduces extra complexity in the system since it is more difficult to find a data set or to maintain a file authorization system over several geographically distributed computers. The purpose of a grid file management system is to overcome these difficulties by providing a unified way to find, access and efficiently transfer the available files and data sets. A distributed file management can provide benefits such as increased data durability by the use of file replication and increased transfer throughput by using replicated data from multiple sources. In the initial phase of AstroGrid-D, a platform that easily can be extended to provide this more advanced functionality is developed.

## 1.1 Scope

The distributed file management will not:

- provide any type of graphical user interface for user file management.
- define the security policies for data sets and files within AstroGrid-D.
- define interfaces for user and group management.

## 1.2 AstroGrid-D Architecture Overview

The AstroGrid-D middleware adopts the concept of Service Oriented Architecture (SOA) in which one or more autonomous services are deployed to bring the expected functionality to the end-user. Each service implements a set of methods defined by an interface contract through which other services and applications can communicate with the service. AstroGrid-D consists of a set of core services that exports a set of interfaces which are combined into the AstroGrid-D middleware. These core services are responsible for basic grid mechanisms

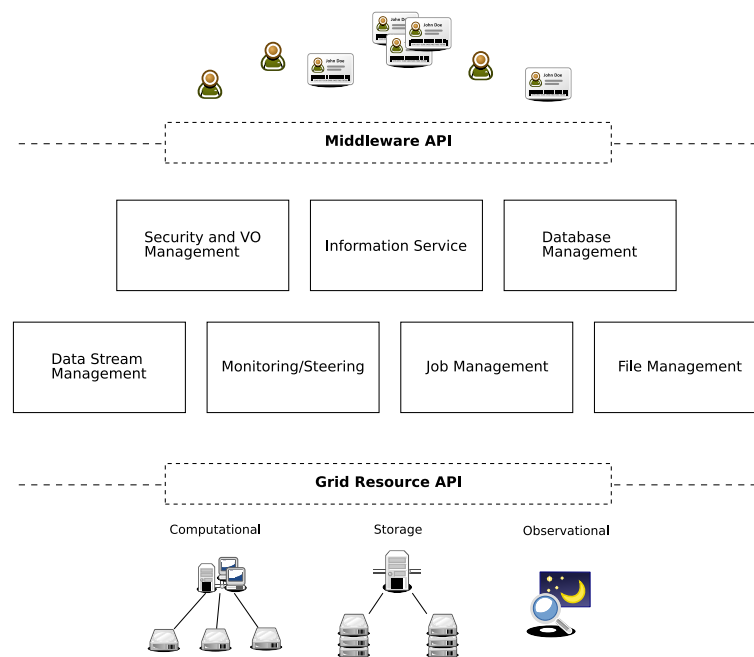


Figure 1.1: A coarse overview of the AstroGrid-D middleware.

such as job management, data management, stream handling, job monitoring/steering and security management. The combined AstroGrid-D middleware can easily be extended by adding additional services.

A service exports a public interface consisting of methods for the service information and service logic. The latter exported methods represent the functionality provided by a service, for example transformation of information or modification of the service state. Service information methods define ways of how to retrieve state information from the service. The information flow can be either pull-based, initiated by the requester, and/or event-based which is initiated by the service after demands made by a requester. The latter type of information flow is also often referred to as publish/subscribe or notification-based information retrieval. Each service also provide mechanisms for authentication and authorization of users. These mechanisms are not part of the public interfaces. Instead, security information is passed within each service request or through pre-initiated sessions using GSI [2]. Furthermore, an AstroGrid-D service can be either stateful, by using the Web Services Resource Framework [11], or stateless depending on the requirements. SOAP [12] should be used for messaging and WSDL [8] should be used for interface definitions. An alternative approach is to use REST [10] principles for service design.

All services (job-, file-, data-, stream-management, monitoring/steering and security) use the Information Service to publish information and to enable discovery of their published information (i.e. a job or the physical location of a file). This is accomplished by letting each service export an interface defining methods for information publishing and discovery. For example, the job management could have an information interface exporting methods to add a new job to the information service and to discover these jobs. The Information Service itself is dependent on the security and VO management service for user authentication and authorization.

Figure 1.1 shows an overview of the most significant services within the AstroGrid-D architec-

ture. The Middleware API is used by a set of users both with and without grid certificates. Users without certificates are restricted to the services and information which are public within the grid. This is to encourage collaboration and to make results easier to publish for certified AstroGrid-D users. At the bottom of the picture are the resources which range from different computational resources, storage resources to observational resources such as robotic telescopes. They interface to the grid services via a resource API containing methods for resource information dissemination, job handling and data handling. Each service and a brief summary of their main responsibilities is presented below. Note that the ordering in figure 1.1 do not imply any hierarchy and flow of information between the services.

**Information Service** - Supports the grid services, applications and application users with functionality for storage and discovery of information.

**Job Management** - Handles job submission and scheduling of jobs based on current information about the grid resources.

**File Management** - Is responsible for file transfers and replica management.

**Data Stream Management** - Handles efficient distributed data stream processing.

**Database Management** - Responsible for database access from the grid.

**Monitoring/Steering** - Provides mechanisms for monitoring of jobs and grid resources and steering of currently active grid jobs.

**Security and VO management** - Manages users and Virtual Organizations within AstroGrid-D and is also responsible for user authentication and authorization.

## 1.3 Document Structure

The remainder of this document is structured as follows. Section 2 provides an overview on the use cases. The requirements derived from these use cases are described in section 3. The following section presents the architecture for the AstroGrid-D distributed file management service. The interface for accessing the service is defined in section 5.

## 2 Use Cases

The members of the AstroGrid-D community have developed a couple of use cases representing typical astrophysical scenarios. Besides a precise description of the current way of use, the use cases also outline how to use a distributed environment, i.e. the AstroGrid-D community Grid.

The use cases were developed in two phases. In the first phase the users where asked to provide an overview of their application, this was mainly to allow a freedom of specifying current and potential future needs. The second phase included a detailed questionnaire that ensured a more comprehensive view of each use case. We have classified the use cases into (A) simulations, (B) data analysis and (C) astronomic observations. In the following paragraphs, for each class, we introduce the main characteristics concerning the management of information.

### 2.1 Simulations

There are six use cases describing simulation runs. A simulation usually expects a relatively small amount of input data, which is processed in an iterative manner. At each simulation step a large number of output data may be generated depending on the problem size and the configuration parameters. Because such simulations perform complex computations they are usually requiring long execution times. Thus, most of the applications already provide mechanisms for observing their progress. Some applications even facilitate on-line steering, e.g. adapting parameters to speed-up the processing.

#### UC1 - NIRVANA

NIRVANA is a grid-based<sup>1</sup> computer code for the solution of the equations of magnetohydrodynamics and for the Poisson equation.

More information on the code can be found on <http://nirvana-code.aip.de>.

As a parallel MPI job, a NIRVANA simulation needs a fixed set of input files to run: an ASCII parameter file to control the simulation; possibly some binary checkpoint files if the simulation should be restarted from an earlier run. All input files have filenames which are known in advance, in a known location on the local filesystem. Their size is problem-dependent, it can be up to 50 GB. Checkpoint files are accessed only once in a simulation.

Each simulation produces an ASCII logfile plus a single or multiple distributed (one per processor) binary output files in HDF5 file format. All files are written to the local filesystem. The estimated maximum (problem-dependent) total sum of output files is 2 TB. A script is used to copy them back to a user-specified location after program termination for post-processing/visualisation purposes.

---

<sup>1</sup>*grid-based* meaning the underlying implemented program algorithm, unrelated to Grid computing

Currently application monitoring is done manually by periodically checking the constantly appended application logfile via direct file access. Access restrictions to data files are not explicitly mentioned.

## UC2 - AMIGA

AMIGA is a code for cosmological N-body simulations. More information can be found on <http://www.aip.de/People/AKnebe/>.

AMIGA is a serial application (parallelisation is planned), simulations can run on single CPUs only. Similar to NIRVANA, an AMIGA job takes an ASCII parameter file (provided by the user) and a single binary file with initial conditions (up to 3 GB, provided by an external program) as input. The initial conditions datafile is usually accessed only once; only for parameter studies (which are not the typical use case for AMIGA jobs) the same initial conditions (but a different set of input parameters) are used for multiple simulations.

Binary snapshots (again up to 3 GB per output timestep, up to several hundreds of output timesteps in total) are written periodically and after program termination post-processed/visualised or used as input for another AMIGA simulation. Optionally some post-processing of the output files can be done by AMIGA on the fly (if requested by the user and switched on during compile time of AMIGA). With this feature switched on, an AMIGA job produces some additional output files (with predictable file names, up to 500 MB per simulation) containing galaxy catalogues as skimmed information from the simulation raw data.

All files are expected to be found on/written to a local filesystem.

`stdout/stderr` messages are written only at startup, further runtime output and progress information are then dumped entirely into an ASCII logfile. Interactive application monitoring is done by periodically checking the contents of this logfile.

Application monitoring via interactive logfile access should be possible by the job owner only, whereas the simulation results shall be made accessible also to other scientists.

## UC3 - NBODY6++

NBODY6++ is a member of a family of high accuracy direct N-body integrators used for simulations of dense star clusters, galactic nuclei, and problems of planet formation.

More information can be found on <http://www.nbodylab.org>.

NBODY6++ simulations are usually run as parallel jobs on supercomputers (eg. submitted as UNICORE jobs to NIC Jülich) or clusters (usually submitted as PBS batch jobs), possibly with special floating-point acceleration hardware (GRAPE boards). They have only moderate requirements on grid file management.

A simulation needs an ASCII parameter file with control parameters to start with (provided by the user) and a data file with N-body configuration either from an earlier NBODY6++ run (in a specific binary format) or from other applications (in a standard format). The data file (maximum size 100 MB) is either copied/moved to a local filesystem from a known external resource or generated in a separate pre-execution step. Input data are only used by one job at

a time.

All output goes to the local filesystem. Diagnostic information is written to `stdout/stderr` which is used to monitor an NBODY6++ simulation. A couple of text and binary files are stored and later retrieved by the user for post processing and visualization. They can be deleted at the computing site and are not needed for restart. Two binary output files are complete dumps for restarting the run. They can remain local, unless the job is to be continued at another location. The total maximum size of all output files can be up to several GB. Output filenames are known in advance, usually some job metadata information (eg. batch job ID plus current date) is used to create them. In the "job farming" computing mode, the same NBODY6++ simulation is run on all processors, with each processor outputting its own files (reflected in the output file name).

The NBODY6++ code is being used and developed at a number of national and international institutes (which are not primarily a part of the AstroGrid-D community). Therefore the architecture of the Grid file management service, as well as the other services developed by AstroGrid-D, should acknowledge this broader application context and provide compatible solutions.

## UC4 - Dynamo

Dynamo is an application for solving the induction equation with turbulent electromotive force (Alpha tensor) for modelling the turbulent dynamo in planets, stars and galaxies.

Dynamo is typically run in task farming mode, i.e. a set of single-processor Dynamo jobs is submitted to available computing resources (either a shared memory system, or multiple machines for which the executables need to be prepared for the given hardware architecture). Each job has a makefile for compilation on the target machine, a user-provided ASCII parameter file and, in case of a restart, a binary input data file (of 0.1 – 1 GB size). All jobs access a different restart file (depending on the parameter settings) and the input data is accessed only once per job. If jobs are restarted or migrated, the specific restart file and an update of the parameter-file is required.

The program writes cyclic binary dumps (0.1 – 10 GB size) at predefined output timesteps. Data analysis of the output is done as a post-processing step where output files are copied back to the user's machine. For long-time simulations a visualization of snapshots is also desired during runtime though.

Information about the progress of the program is written to `stdout/stderr` and an ASCII file. Interactive monitoring is done by the job owner by periodically checking `stdout/stderr` and the ASCII logfile.

Since each job keeps all data (input and output) in a specified directory (known in advance) on the local filesystem, the datafile transport to/from the execution hosts can be fully automated in both directions (input and results) in a future Grid environment, along with the automatic generation of the executable.

## UC5 - Cactus

Cactus is used by the physicists in the Numerical Relativity department of the AEI to numerically simulate extremely massive bodies, such as neutron stars and black holes.

More information can be found on <http://www.cactuscode.org>.

Cactus simulations are typically run as parallel MPI jobs on clusters and/or supercomputers. Requirements on the amount of available disk space are quite high: apart from temporary space for creating intermediate checkpoints, the size of simulation output datafiles are usually large, depending on the size of the simulation (how many processors), the total runtime (how many timesteps), and the I/O parameters specified by the user (which variables to output how often in what format).

A Cactus simulation requires an ASCII parameter file as input which is provided by the user. In case of a restart from an earlier simulation, a checkpoint to recover from is needed as well. Each checkpoint itself may consist of a set of binary files in HDF5 file format; the total size of the checkpoint is in the order of several hundred GB (it is assumed that a checkpoint contains the entire state of the current simulation at a given time, and therefore matches the amount of allocated main memory in the maximum case).

The names, number and size of output files per simulation is highly dependent on the I/O parameters used by that run. Since many of these parameters can be steered at runtime, it may not be known in advance which files will be created when. Usually the user specifies a main output directory in the parameter file, into which output is then written directly or into subdirectories thereof. By default, both ASCII and binary datafiles are written, the former being rather small, the latter (written as one file per processor) ranging up to several hundred GBs.

Each job's `stdout/stderr` messages are stored in intermediate logfiles determined by the local queuing system (eg. in a local PBS spool directory located on a specific compute node, therefore not directly accessible from the headnode!); in addition, application-specific log- and ASCII datafiles may also need to be accessed during runtime for interactive monitoring. While the latter is achieved through existing monitoring functionality integrated in the simulation code (thorn HTTPD), access to temporary `stdout/stderr` logfiles is done via a proprietary shell script (which determines the first compute node of the job, opens a remote shell on that node and fetches the most recent contents of the corresponding logfile in the PBS spool dir).

Although Cactus provides means for online visualisation of intermediate simulation results through integrated monitoring and visualisation services, detailed data analysis is typically done as a post-processing step. Due to limited disk space and/or network capacities, it is in most cases not feasible for the entire set of simulation output files to be staged to some local machine and then do the analysis there as usual. There are two approaches for Cactus data analysis in a Grid environment: (1) Full data analysis is performed as before on the entire set of output data but now needs to be executed on the resource where the data resides (ie. where it had been generated). (2) For a preliminary data analysis of a Cactus simulation run it is often not necessary to access the entire set of data but only take a few selected samples (eg. a certain group of variables from a specified number of timesteps). Such data sample analysis can then still be done as before on a client's machine but requires efficient (meaning: partial) access to remote files (such as HDF5 data files).

## UC6 - GADGET

GADGET is a freely available code for cosmological N-body/SPH simulations on massively parallel computers with distributed memory.

More information can be found on <http://www.mpa-garching.mpg.de/gadget>.

When starting a GADGET job, the user provides a startup parameter file and 1-3 binary files with initial conditions (this has been generated prior execution). The initial file must be readable by all nodes taking part in the simulation. Depending on the number of simulated particles, the size of the initial conditions file can range up to some GB. The number (runtime/snapshot-time) of binary snapshots, which freeze the state of the simulation at a given time, is configurable in the parameter-file. The size of a snapshot is roughly given by the size of the initial conditions. Thus the total output of a job is fairly well estimated by (number of snapshots)\*(size of init-file). Snapshots are migrateable. Additionally, restart-files can be generated, which freeze the state of memory/processor. These are not migrateable. Optionally, a binary snapshot can be specified in the parameter file for restart from an earlier simulation run. The size of a snapshot can vary from 40 MB up to 20 GB, again dependent on the number of particles. While the input data is generally small (except for starting from a checkpoint), the output volume ranges from 100GB to several TBs per job.

Each snapshot can be written in parallel by dedicated I/O processors (ie. every 4th processor gathers its own state and the states of the other 3 associated processors into a separate file). After being written, a snapshot can be transferred to some local machine and visualised with IDL, PMViewer, or other graphics programs). They can also serve as restart files for future GADGET simulations. Input and output files are generally read from/ written to either a local disk or some NFS-mounted storage.

Beside runtime information on `stdout/stderr`, each GADGET job also produces a set of ASCII log files for evaluating code performance (`cpu.txt`), showing the number of performed timesteps (`info.txt`), the work load and particle load balance (`timings.txt`), or controlling the energy-balance (`energy.txt`). In a Grid environment it should be possible to interactively access `stdout/stderr` for monitoring purposes, and to visualise the status information in the other ASCII logfiles and access the snapshots already written to disk.

## 2.2 Analysis of data sets

There are six use cases describing data analyzation runs. The data sets are usually very large (up to Terabytes). Because of limited network bandwidth and traffic volume it is necessary to reduce the amount of data which is transmitted from the data storage (file server or database) to the compute resource where the analysis is performed. While some scenarios explicitly request the movement of the data, others consider the movement of the analysis code to the data storage for performing the operations on them (provided that the data storage provides sufficient compute power). The size of the results is usually smaller than the size of the input data. Further improvements may be achieved by collecting information about data provenance. Thus already generated data products may be fetched from a storage rather than generating them again and again.

## UC7 - Astrometric matching

Astrometric Matching for classification of Spectral Energy Distributions (SED) is an essential research technique to discover new astronomical objects like obscured active galactic nuclei (AGNs), brown dwarfs, isolated neutron stars, or planetary nebulae (PNs).

The data about the astronomical objects is usually stored in databases, called catalogues. Besides a list of catalogues the information service could store information about the schema and the characteristics (e.g. volume, version number) of the data objects. For query optimization the scenario uses several static and dynamic information about the computing, storing and networking resources. The static information refers to the maximum capabilities of the resources, for example processor speed, disk capacity or network bandwidth. The dynamic information covers data on the current usage of the resources, for example processor load, available disk space or network traffic. In addition information about the software environment of a resource (in particular for computing resources) is required. If the output data is held for further processing then also the schema and characteristics of it could be stored in the information service.

## UC8 - Clusterfinder

The purpose of clusterfinder is improving the source-identification of X-Ray galaxy clusters by correlating data of X-Ray photon maps (ROSAT All Sky Survey - RASS) and optical galaxy maps (Sloan Digital Sky Survey - SDSS).

More information can be found on <http://www.g-vo.org/portal/tile/products/services/clusterfinder/index.jsp>.

The core of the application is a Fortran program, whose first action is to read in a parameter file containing tuning parameters, filenames for mask maps and the pointspread function, ra/dec-ordered values of catalog data, and I/O files. The map-files, pointspread function file and the catalog data file are read by the program before the calculation starts. The calculated likelihoods are stored in an output file. A input data set may be processed multiple times in a short period of time. The results may be used in a post-processing / visualization step.

The total size of the input data ranges from 10 MBytes to 100 MBytes. The size of the output file (likelihood map) ranges from 1 MByte to 50 MBytes.

## UC9 - Virtual Telescopes

In “Theory VO” (VO=Virtual Observatory !) parlance, a “Virtual Telescope” is a piece of software that mimics an astronomical observatory (telescope/instrument/satellite/spectrograph etc.) which can “observe” the result of a simulation and produce a synthetic observation that can be directly compared to the output of the instrument that is being simulated.

In this scenario we propose to make available a large set of simulation results of galaxy clusters together with a virtual telescope that simulates an X-Ray satellite. Users will be enabled to search a database for simulations of interest to them, which are stored somewhere on the data grid. They can also find a virtual telescope (grid) service that can be applied to the selected simulation result. They can then execute the observation somewhere on the grid

where sufficient compute and data storage resources are available to them. The results will in general be relatively small images that should be returned to the user.

The input data is provided in a configuration file. Its size is approximately 500 KBytes. Because single data sets may be used several times, they may be replicated to avoid unnecessary data transfers. The output data consists of multiple files – a data file (FITS or JPEG format), logging files and parameter files. The total size of all output files is about 1 MByte.

### **UC10 - Millenium Query**

The scenario Millenium Query is about selecting a subset of a large remote dataset and moving the result to the user. The input data is stored in a relational database system. Hence it is not handled by the file management component. The output data can be stored in a database or in a file. For the latter, the file management just needs to provide methods for adding them to some storage server, discovering them (via the AstroGrid-D information service) and for retrieving them from a known location such as a GridFTP or HTTP server. The size of the output data is upto 1 GByte. The file access methods may require the use of standard access permission checking.

### **UC11 - Simulation Post-processing**

The scenario Simulation Post-processing deals with on-demand post-processing of the output data from cosmological simulations, in particular, but not restricted to, the 25 Terabyte Millenium Simulation run of the Virgo consortium. One of the more significant challenges is the management of the output data from simulation runs and the post-processing.

A post-processing job is often executed using the ProC in conjunction with the DMC 2.3 which handles the data logistics. The input to a post-processing job can be parts of the Millenium Simulation or any of a smaller set of simulations also produced by the Gadget simulation package. Typically, a job processes between 20-100 data sets which are from 10 to 20 GBytes of data. Each data sets consists of several files with size between 1 MByte to 2 GByte. In total, a post-processing workflow will process  $10^2$  to  $10^5$  files. Currently, the input data format is a special Gadget format, but alternatives such as HDF5 is investigated. The same input data may be accessed several times during a post-processing run. Output results are typically much smaller than the input and are transferred directly to the user after the run is finished.

### **UC12 - GEO600**

The gravitational wave detector GEO600 near Hannover and other ground-based devices (eg. LIGO in the U.S., TAMA in Japan, VIRGO in Italy) aim at the direct detection of gravitational waves by means of a laser interferometer. The detectors constantly generate data (eg. LIGO in the order of 1GBytes/day) that needs to be searched for gravitational wave signals (eg. by filtering the raw input data, running various transformations and check for signals).

All input data is stored in a pre-known location shared via NFS. Data is accessed using the FRAME I/O library in addition to POSIX based I/O. The size of data can range from 40 MBytes (one data unit) to 10 GBytes depending on the number of timesteps that are analyzed.

Each data unit can be accessed multiple times. Also, the scenario is executed several times with the same input data, but with different input parameters. Output data is written directly to the disk of the worker node and is later staged manually by the user. The size is typically much smaller than the input data and is later used to produce graphical histograms using Matlab scripts and OpenDX networks.

The GEO600 scenario can use a grid environment in several ways. More compute resources can be used by running parameter studies on multiple grid resources. It is also possible to parallelize a single scenario since each data unit is analyzed independently of the others. Output results are made available to a broader public using a grid portal.

## UC16 - FRINGE-MIDI

The German Center for Infrared and Optical Interferometry called FrInGe coordinates efforts by German institutions in obtaining, reducing, and interpreting astronomical interferometric data from optical to mid-infrared wavelengths. Currently, interferometric efforts in Germany concentrate on instrumentation for the Very Large Telescope Interferometer (VLTI) (especially MIDI and AMBER), the Large Binocular Telescope (LBT) interferometric capabilities (especially LINC), and on contributions to the planned space interferometer DARWIN.

Mid-infrared (MIR) interferometry at the ESO site on Paranal is performed with 8m telescopes as well as relocatable outrigger telescopes of 1.8 m diameter. MIDI, the mid-infrared interferometric instrument, is currently operating two 8m telescopes in the wavelength range from 8 to 13 micron. The follow-up instrument MATISSE, the Multi AperTure mid-Infrared SpectroScopic Experiment, is foreseen to be able to combine four telescopes at the VLTI site in interferometric mode.

With the VLTI being the largest array of 8m telescopes and with the unmatched spatial resolution of interferometric observations, the data obtained with the MIDI instrument represent the input for typical high-end astronomical data reduction.

In this usecase, the raw MIDI data obtained at the VLTI are fed into the Interactive Data Language tool IDL using the MIA-EWS - MIDI Interactive Analysis - Expert Work Station package which also makes use of routines written in C.

The challenge of UC16 is the possibility of a complex interaction of the user during the data processing. We will start with a black box application that produces so-called *visibilities* from the raw data. Realistic applications, however, will have to be designed to incorporate user interactions changing and triggering new work flows of data reduction. The size of the current raw data is several ten GB.

The visibilities, in turn, will be target of a post-processing with radiative transfer (RT) tools modeling the source at all observed wavelengths. 3D RT programs operate in Fortran and have large memory requirements (several GBytes) due to the high-dimensional solution vector. The runtimes of 3D RT programs are weeks on a single high/end processor.

Raw data must be staged from the initial data provider (ESO) to MPIA. When available at MPIA, the data is analysed automatically, using a program, or by a scientist. This analyzation results in a set of visibilities. They are typically much smaller than the raw data and can easily be transferred to other locations. Visibilities are then used for computationally expensive post-

processing. The same visibility can be accessed several times and post-processed by different tools, including the previously mentioned 3D RT programs. Additionally, more visibilities are added over time, when available from the ESO, to refine the output results.

## 2.3 The Planck Process Coordinator (UC13)

The Planck Process Coordinator (ProC) is a workflow engine originally developed for running simulation and data analysis workflows (called “pipelines” in astronomical parlance) that occur within the Planck Surveyor project. Because it can be used for both simulation runs and data analysis (or even both combined) we did not assign it to one of the previous categories discussed in Sections 2.1 and 2.2.

The ProC is tightly coupled with a metadata and data management component (DMC). This component stores metadata assigned to a data object and, depending on configuration, the data object itself. Both intermediary workflow results and the final results are stored and accessed via the DMC. Integration of the AstroGrid-D file management and the DMC is currently not foreseen for the close future. Since the DMC and the AstroGrid-D file management and information service provide similar functionality, the ProC can, with some effort, use these grid components.

## 2.4 Integration of Robotic Telescopes

The partners maintain or have access to robotic telescopes. These robotic telescopes can be seen as “normal” Grid resource similar to a compute resource. Instead of executing a compute job they perform observations which also produce data. Besides more complex descriptions of jobs, the management of robotic telescopes also differs in that most observations must be performed during certain time windows. Then also the local weather conditions need to be taken into account for Grid-wide job/observation distribution.

### UC14 - STELLA

The AIP maintains a robotic telescope at the Tenerife Island. The telescope should be seen as a “normal” Grid resource. Advanced usage scenarios, such as instant reaction to astronomical events or arbitrary long observation of specific objects, become possible with a Grid of world-wide robotic telescopes.

Input data consist of an XML-file with the configuration parameters of an observation run. This file is very small and is transferred manually to the telescope control computer using scp. The STELLA telescopes (I + II) produces 8 MB resp. 32MB per observation. This data is stored at a local 1 TB storage and transferred to an archive at AIP. The local storage is cleared as soon as the data is secured at AIP. Output data is stored in the FITS file format, which also include metadata describing the file. Additionally, the access to scientific data is restricted to a certain group of users and belongs to the Principal Investigator for three years after the observation.

The ultimate goal for the STELLA use case is the possibility of linking multiple robotic telescopes with the result of better reaction for the end user and arbitrary long observations of a single target. The grid file management in conjunction with metadata describing the results can simplify the management of the geographically distributed output data.

## 3 Requirements

### 3.1 User Requirements

The following requirements are specific to the distributed file management in AstroGrid-D. They were derived from a set of Use Cases, summarized in section 2. The numbering of user requirements is sequential with the prefix "UIS". Note that this list is in no way exhaustive, but a subset of the most significant requirements.

**UIS1** Automated staging of input and output data (all UCs)

Before and after application activity. This includes staging of parameter-files, checkpoint-files and input data. After an application is finished it is assumed that output files are staged to a location, both specified in the job description, and removed at the execution site after successful transfer.

**UIS2** Monitoring of `stdout/stderr` log-files (all UCs)

Grid jobs started on a cluster by a local queuing system get their `stdout/stderr` streams redirected to intermediate log-files. These log-files may be stored on "hidden" nodes. A "hidden" node is a worker node behind the cluster's firewall and/or within the cluster's virtual private network, with access only through the frontend.

At present such `stdout/stderr` log-files are accessed by the user by logging in directly to the worker node from the frontend. Since this type of remote access may not be available for all users at all grid resources, there must be an alternative way of accessing these log-files.

**UIS3** Interactive access to intermediate files (UC2, UC4, UC5, UC6)

Files generated by the application should be accessible during run-time. In addition to UIS2, applications running on clusters can also write their own log-files on "hidden" nodes. Again, these files must be made available to the user for application monitoring purposes by the grid file management service. Access to this type of files implies a requirement for partial access to the last part of the file, since it is inefficient to retrieve the entire file at each access.

**UIS4** Remote partial access to HDF5-files (UC5)

When simulation results should be postprocessed and/or visualised, it is often not feasible for large output files to be staged as a whole from the data source (e.g. a filesystem on a cluster) to the postprocessing machine (e.g. a local visualisation workstation): the local machine does not have enough disk space available to store the output files and/or the network connection is too slow to transfer several hundred gigabytes within a reasonable amount of time. Therefore large simulation output files (eg. Cactus HDF5 datafiles with multiple output variables and many simulation timesteps) should be accessed remotely, fetching only those parts of the files which are to be visualised at the same time (e.g. a

subset of variables of a given timestep).

If possible, the necessary functionality to implement partial access to remote HDF5 files should be transparent to existing postprocessing applications (reuse standard interfaces and services).

**UIS5** Provide access to files for both AstroGrid-D members, identified through a valid AstroGrid-D X.509 certificate, and non AstroGrid-D members, everyone without AstroGrid-D X.509 certificate. (All UCs)

**UIS6** Restrict access to a file or a set of files. (All UCs)

## 3.2 Grid and Generic Requirements

These requirements are not specifically stated but are more implicit assumptions when using a grid file management system. The numbering of grid requirements is sequential with the prefix "GIS".

**GIS1** It must be possible to insert files into the system.

**GIS2** It must be possible to remove files from the system.

**GIS3** The system must provide discovery and registration of file replicas.

**GIS4** A data transfer must be traceable (possible to monitor).

**GIS5** There must exist a mechanism to ensure file integrity, i.e. ensuring that a file is complete and un-changed after a transfer.

**GIS6** Third party transfer

Basically, third party transfer is initiated by a client, instructing a file transfer between two sites part of the file management. It is necessary when, for example, the job management need to instruct the file management of staging a file to the site of job execution.

## 4 Architecture of the Distributed File Management

Grids are cooperative environment for sharing of computational resources, but sharing of storage and network resources are equally important in a Grid environment. AstroGrid-D consists of several compute resources with different storage capacity for grid-job input and output files. Depending on locally defined policies, data needed for the execution of a grid-job is treated differently. For example, some sites does not have the possibility to store job output data after a job has finished. Another case may be that the site removes job output data after a pre-defined time period starting from when the job finished. Thus, automated pre- and post-job staging, movement of data to and from the compute site, is necessary to comply with local policies. Furthermore, when a compute site is participating in one or more grids, it may get a sudden increase in the number of potential users. Grid middleware deployed at the site must acknowledge this and use the local resources efficiently. For example, a job's output data should be cleaned when it is not in use any more.

When staging data, it is assumed that both the data originator and receiver are available during the entire data transfer. However, since many interactions with the grid middleware are asynchronous, the host containing the input files or is assumed to store the output files may not always be available at the time of transfer. For example, a user uses her local laptop to store input files and submit jobs to the AstroGrid-D job management. Later on, when the file management has received the staging request from the job management or the compute site, it will try to access the laptop with the input files. This may then fail, with the consequence that the job fails. One reason can be that the laptop is behind a firewall or that it was disconnected because the user left for the day. A solution to this problem is to introduce an intermediary storage, responsible for storing job input and output files until the job has finished and the user have had the possibility to transfer the output files to persistent storage.

From a user perspective there are several benefits for an intermediary storage. For example, management of distributed data, easier sharing of large data sets and aggregation of existing storage resources.

**Distributed data.** Files within a grid environment become distributed to several storage resources over time due to different job and resource constraints such as storage capacity and resource workload. Finding these files without a unified file view is often difficult since different storage resources do not share a common structure. Additionally, maintaining files within a group of distributed users is more complex than the single user scenario. A user can, for example, move a file without the knowledge of the group. The storage layer include data in a common namespace, making data movement between distributed resources transparent to the user. In order to easier find data, the user can assign application-specific or scientific metadata to data. The AstroGrid-D Information Service (IS) [13] assists the user in storing and querying metadata.

**Data sharing.** Data can be made publicly accessible through the storage layer directly by a researcher. This makes it much easier for other researchers, for example in collaborations, to access the data.

**Storage and network capacity.** The amount of data that can be stored in the storage layer exceeds the capacity of a researchers local storage capacity. This can be useful to provide large or large amount of files to the community. Furthermore, by making data available on multiple locations, by creating replicas, it is possible to save bandwidth at the site where the data originated. Replicas are also used to increase the availability of a data set.

An important observation is that normal workstations and desktop computers used in the network environments of corporations and academic institutes often have large amounts of free storage [7]. Additionally, the clusters and workstations used in AstroGrid-D contain storage capacity. The storage layer consist of voluntarily shared storage capacity from both AstroGrid-D users, i.e. a user with a valid AstroGrid-D certificate, and institute resources. Aggregating these assets into a storage layer fits well into the grid vision of sharing resources.

The distributed file management in AstroGrid-D is not only responsible for the storage layer. The following are additional goals for the distributed file management.

- Aggregate the available storage capacity in AstroGrid-D resources.
- Provide a unified view of data stored explicitly stored in the AstroGrid-D storage layer.
- Allow for efficient partial access to data.
- Stage data and monitor data transfers required for executing grid-jobs.
- Interactive access to log-files and results.

The last three goals have been identified earlier by the grid community and are solved by existing software components. The technical infrastructure for the distributed file management will include Globus Toolkit 4 (GT4) and GridFTP to support automated data staging, interactive access to intermediate files and remote partial file access. For user requirement UIS4 (partial access to HDF5-files), GridFTP with a special plugin is used to accomodate partial access to HDF5 files. However, while AstroGrid-D include mainly cluster and workstations, many of the distributed storage grid-solutions available today are aimed at HPC and cluster systems, for example, SRB [5] and dCache [1]. They also have a high administrative overhead for handling dynamic situations with nodes joining and leaving the distributed storage. The rest of this chapter describes the AstroGrid-D storage layer design. Additionally, special requirements for grid nodes are presented together with the security model for files and how to associate metadata with files in the storage layer.

## 4.1 Storage layer overview

Figure 4.1 shows an overview of the storage layer. The lowest part of the picture depicts AstroGrid-D storage resources in different Internet-connected domains. A self-organizing overlay network, used to connect storage resources, is formed on top of the storage resources. A

storage resource part of the storage layer is called a storage layer node or just node for simplicity. Each node store files and metadata about files. Additionally, each node maintain connections to other nodes in the overlay network. These connections are structured to create a one-dimensional index structure used to locate metadata about files, for example, physical location or creation time. The interface separating the storage layer from the clients include operations on files, such as post and get, but also operations for metadata lookup.

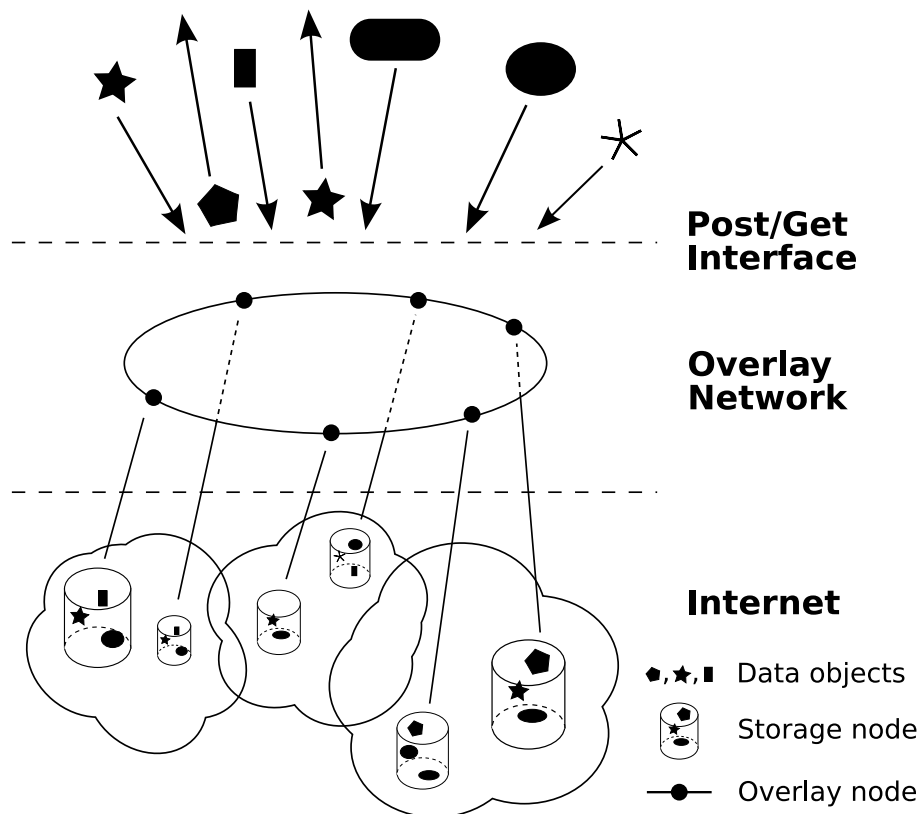


Figure 4.1: Overview of the storage layer.

Three components are identified, forming the core of the storage layer:

**Storage layer node.** A resource responsible for providing storage. Each node implements the external interface used by storage layer clients.

**Data object.** A data object contain data from a file and metadata about the file. File data does not need to be stored together with the metadata. The storage layer store data objects on storage layer nodes.

**Structured overlay network.** A network consisting of storage layer nodes operating on top of the Internet. Provides functionality of a one-dimensional index structure used for storage and efficient lookup of system-specific metadata.

From the requirements and the scenarios described earlier, the following goals have been outlined for the design of the storage layer.

**Common namespace.** Jobs in AstroGrid-D will be executed on different compute resources often not decided by the job issuer. The consequence is that the user does not have control of the location of output and input data. A common namespace can remedy this situation by assigning globally unique identifiers to each data set, creating a layer of indirection used by the user to find the real data location.

**Deployment and self-management.** Since the aim of the storage layer is to use voluntarily shared storage capacity it is important that there is a low deployment threshold of the software. Furthermore, the resources where the software will run is not always operated by experienced administrators, why the continuous maintenance must be negligible. It is also vital that the software considers a moderate amount of deployments joining and leaving the storage layer at any time, due to the usage of normal workstations, unintentional hardware failures and downtime.

**Integration.** Compatibility with the grid software used in AstroGrid-D (GT4, GridFTP, etc.) is important for the integration of the storage layer into the middleware. Therefore, standard transfer protocols such as HTTP and techniques for single-sign-on with X.509 certificates should be used.

**Fault-tolerance.** There must exist mechanisms handling hosts leaving the storage layer prematurely, i.e. without explicitly notifying the storage layer. Sudden node departures can for example happen because of hardware or software failures.

**Resource fairness.** Since the storage layer nodes in many cases will be normal workstations and servers used for other tasks, it is important that the storage layer leaves a small footprint in CPU, maintenance bandwidth and memory usage.

## 4.2 Data objects and operations

A data object contains metadata describing a file and how to retrieve the file's data. Each data object has an address, called Object Identifier (OID). The OID is used to access a data object in the storage layer. An OID consists of a user specified namespace and a user specified path. The user specified namespace separates the name of different data objects from different users and applications. For example, the user "Pelle Test" creates a namespace "pelle", where he stores his data objects. While a group of users using the application "cactus", decides to store all their data under the namespace "cactus". The user specified path correspond to a file name within a namespace. The OID naming schema creates the common namespace mentioned earlier. Naming conflicts of the user specified namespace is ignored, instead a First Come First Serve (FCFS) heuristic is applied, where the user first adding data objects within a namespace is responsible for it. FCFS is sufficient since the user-base in AstroGrid-D is small.

Metadata stored by a data object include a list of URIs, representing physical locations of the file data. The OID is used to retrieve the data object metadata from the storage layer. Replicated data stored as part of the storage layer or copies of the data stored outside the storage layer are registered by adding a new URI to the list. Registration of external data is necessary to include files that are too large to be stored in the storage layer, but still should be part of the common AstroGrid-D namespace. However, applications using this functionality must themselves assure that the data is accessible. Moreover, to increase the consistency of

accessing externally stored data and to accommodate garbage collection mechanisms needed for fair usage of the shared storage capacity, each data object is assigned a Time-To-Live (TTL) value. This value indicates how long the data object is stored before it is removed from the storage. The TTL-value must be updated to avoid removal.

Any storage layer node have capability to return any data object registered in the storage layer, making data accessible independent of its location. Additionally, replicated data objects are accessed transparently by location URIs from the data object metadata. Another advantage of a common namespace is that it is easy to associate RDF-based metadata to a data object by prefixing the OID with a base URI. This base URI should be independent of the actual storage location, since the data object will reside on different storage layer nodes over time. Note that an OID prefixed with the base URI does not need to be network accessible, i.e. directly accessible by a client. For example, this metadata could include provenance information such as the name of the simulator or data analysis tool that created the data object or values from a FITS-file metadata header. Section 4.7 introduce different strategies for assigning metadata to an OID.

In order to increase the fault-tolerance of a stored data object, an archived version is created by dividing the complete file into many smaller blocks, enough to re-create the original file [9, 6]. Archived data objects are immutable, meaning that subsequent changes to the data object is not allowed. Each data block has a Block IDentifier (BID), derived from the content of the block using a function  $H_{BID}$ . All BIDs are stored in a data structure part of the data object's metadata. Thus, the list of BIDs are retrievable by using the OID. A data block is stored on a storage layer node assigned to be responsible for the BID. While this technique requires a small overhead in storage capacity, it increases the fault-tolerance since blocks are located on multiple nodes. Furthermore, since blocks are located at different nodes they can be accessed in parallel. This may lead to increased throughput when transferring a data object within the storage layer or when archived data objects are extracted from the storage layer.

Updates to a data object is possible until an archived version is created. When updating a data object several different versions of the object exist in the system. Hence, the storage layer must implement a locking scheme applied to the data object's replicas in order to avoid inconsistent or old data retrieved by clients. Since updates to replicas are a source of inconsistency, it will not be considered in the early versions of the storage layer and, as a consequence, data objects with replicas are considered immutable. However, if a data object does not have any replicas, the user is free to update the data.

## Operations

There are six operations on data objects: insert, retrieve, remove, replicate, archive and register replica. Security-related operations are covered in Section 4.6

**Insert.** Add a file to the storage layer. This creates a data object including metadata.

**Retrieve.** Transfer file data from the storage layer to a local storage.

**Remove.** Remove a data object from the storage layer.

**Replicate.** Creates a replica of the file data for a given data object on a specified node within the storage layer.

**Archive.** Creates an archived version of the specified file object.

**Register replica.** Adds an external location of file data associated with a data object. This means that the file is not physically stored on a storage layer node.

### 4.3 Storage layer nodes

A storage layer node is voluntarily sharing storage capacity and bandwidth to the storage layer. The storage capacity is used to store data objects, while the bandwidth is used to maintain the storage layer and transfer data. A node implements different mechanisms to manage the local data and both externally and internally available interfaces to give access to the data. Nodes are members of a structured overlay network, used to handle storage layer node joins and leaves and to provide decentralized storage and lookup of data object metadata.

#### Node Identifier

Each node has a unique Node Identifier (NID). The NID is generated independently from other nodes by relying on a hash function  $H_{NID}$ .  $H_{NID}$  takes the hash-value of the hosting user's X.509 certificate and concatenates it with a hash-value of the host IP address and port on which the node instance is running. First, by using the hash of the X.509 certificate, NIDs will be distributed uniformly over the identifier space. Second, the IP-address and port is used to allow the user to run multiple nodes, for example, on different machines in a cluster.

#### Internal and External Interface

All storage layer nodes implement and export the operations on data objects. Additionally, each node implement lookup of data objects based on the OID.

There are two different aspects of integration with grid mechanisms, mentioned as one of the design goals earlier. First, by using file transfer protocols such as FTP or HTTP to access files stored on a storage layer node it is possible to use already existing software for retrieval and staging of files. Second, authentication is done by using X.509 proxy certificates and transport layer solutions based on GSI [2], common in the grid community. Authorization of access to a data object is defined by users and enforced by the storage layer node that store the data object.

Third-party transfers, a transfer between site A and B initiated from a client C, and monitoring of such are an important part of the grid infrastructure. They are especially useful for the job management when used in the staging process. The storage layer supports third-party transfer via the copy operation. Each third-party transfer is assigned an identifier, which can be used to monitor the current state of the transfer. A copy of a data object D to host B is simply implemented as a download from host A. From the storage layer point of view this is equivalent to a replication of D.

## Indexing and metadata storage

Existing data on a node is added to the storage layer by an indexing mechanism. Indexing enables integration of existing data archives and data sharing by individual scientists. It is also used to index new files added to the storage layer. The indexing algorithm maps a file to an OID, creating a data object of the file for insertion into the storage layer. The index is stored locally in an instance of the Information Service developed by AstroGrid-D Workgroup 2 [13].

Many file-formats include metadata as part of the file. An important feature of the indexing mechanism is to extract and add this metadata to the local instance of the Information Service. The indexing algorithm includes a dispatcher for external programs that harvest metadata from the common file-formats used within the AstroGrid-D community (FITS, Frame, HDF5, ...). A harvester first extracts data from the file-format and then translates it into RDF/XML, used as input for the Information Service. It is expected that the community will provide the harvesting programs. Furthermore, the indexing algorithm is used to apply system metadata to data objects. For example, a TTL-value or a flag indicating that the file should be archived.

## Storage management

Management of node storage include mechanisms taking local decisions in order to use the storage resource without unnecessary overhead. This includes deploying an algorithm for load-balancing and a storage negotiation protocol. It also includes scheduling of maintenance of the data object index and garbage collection. Additionally, these mechanisms are necessary to provide a self-managing storage layer node. Two of the design goals are considered here: Resource fairness and Deployment and self-management.

**Load-balancing.** Typically, data objects have different popularity, needing more bandwidth with increasing popularity, and requires different amount of storage capacity depending on their size. A load-balancing algorithm ensures that the used storage and bandwidth is not straining the capacity of the local node. This is done by comparing the load with other nodes and re-locating objects when necessary. The objective function of the load-balancing algorithm aims for an even load over all nodes in the storage layer. See [14] for further details.

**Storage reservation protocol.** When transferring data objects to a node in the storage layer, the requester must make sure that the receiving node has sufficient storage capacity to host the data object. For this, a simple reservation protocol is implemented. A request capacity on B, if B has the capacity it reserves the storage space. If the storage space is not used, B releases the lock after a time-out. The time-out is used to un-lock reserved storage if A fails the transfer.

**Data object index maintenance.** As mentioned in the previous sections, there are several reasons for the data object index to become inconsistent. Therefore, a maintenance algorithm is run periodically to detect and fix possible inconsistencies. Heuristics for the algorithm was discussed earlier in this section.

**Garbage collection.** The garbage collector is run periodically and is responsible for removing

data objects with an expired TTL-value. A TTL-value of infinity means that the data object should not be removed by the garbage collector.

## 4.4 Overlay network design

The storage layer is designed to include everything from clusters to normal workstations. A structured overlay network is used in order to minimize the management overhead when nodes are joining or leaving the storage layer. Furthermore, due to how the connections between nodes in a structured overlay network are designed, it also provides a one-dimensional index, also known as a distributed hash table. This index is used for lookups and storage of data object metadata. Another advantage of a structured overlay network is that it is completely decentralized, avoiding the single-point of failure existing in many of the current solutions, such as SRB. However, self-management induces extra use bandwidth usage and a more complex implementation.

On top of the structured overlay network a distributed hash table (DHT) is implemented. It has two simple primitives: `put(key, value)` and `get(key)` returning the value corresponding to the key. The storage layer is built on top the DHT abstraction. This section discusses how the data objects are mapped into the DHT and how the storage layer handles churn, the dynamic behaviour of nodes joining and leaving the network. First, an introduction to the basic building blocks of a structured overlay network.

**Identifier space** Both identifiers for nodes and for key's stored in the DHT share a common numerical space. In this case, it is an interval  $[0, 2^n[$ , where  $n$  is a large number. Increasing the size of  $n$ , increases the potential number of nodes and data objects of the storage layer. Another reason for increasing  $n$  is that the probability of identifier collision decreases with a larger value.

**Routing table** Each node in a structured overlay network maintains a routing table. The routing table contains other nodes identifiers, called neighbors. There are two type of neighbors, short-range and long-range. Short-range neighbors are numerically close in the identifier space, while long-range neighbors are distributed over the identifier space using an implementation-dependent algorithm. For example, the long-range neighbors can be exponentially increasing in distance [18] or create a prefix-tree [17] starting from the local node. The reason for structuring the long-range neighbors in such manor is to achieve efficient routing of messages.

**Lookup** A lookup in a structured overlay network is key-based, meaning that given a key from the identifier space, the node responsible for that key is found. By finding the node, the correct value corresponding to the key can be returned. This is vital functionality for the DHT abstraction. For this to work, each node must be assigned a key-range, i.e. the range of identifiers the node is responsible for. The key-range can, for example, be all identifiers from the local node identifier up to, but not including, the sub-sequent node's identifier. A key-based lookup in a structured overlay network is typically  $O(\log N)$ , where  $N$  is the number of nodes. Thus, it scales well with an increasing number of nodes in the network.

**Self-organization** Churn, or the dynamic process of joining and leaving nodes, create inconsistencies in the routing tables of nodes in the overlay. Since the routing tables are used for key-based lookup, they must be correct in order for a key lookup to return the correct value. Therefore, each node implement algorithms for node joins and leaves. In addition, each node execute a periodical maintenance algorithm that detects if any neighbor has failed and left the storage layer without notifying it's neighbors.

The metadata of a data object is stored and managed by the DHT. Lookup of data object metadata is done via the OID. An OID is mapped to the DHT's identifier space using a mapping function. This identifier is internal for the storage layer and irrelevant for the user since they access data objects via the OID. The DHT complements the local index at each storage layer node by making data objects globally searchable within the storage layer.

When a new node joins the storage layer it first joins the structured overlay network and thereby also contributes to the DHT. Moreover, all data index by the node is added to the storage layer by inserting the data object metadata into the DHT. Any changes to the metadata, addition of new replicas for example, is updated in the DHT by re-inserting the metadata.

Nodes leaving the storage layer or failing without giving notice is more difficult to handle. The main problem is that when a node leaves, the metadata stored by the DHT contain invalid information. For example, if the metadata for location of the file data is pointing to a physical location that is not accessible any more, the user will not be able to retrieve the data. One assumption used by the storage layer is that failing nodes will return sooner or later. Thus, if a node is only in-accessible for a short period, the metadata that it has registered does not need to be removed. Instead, all metadata inserted in the DHT has a TTL. Each node has a mechanism traversing the locally stored DHT metadata and removes all entries where the TTL is expired. Non-failing nodes use the same mechanism since it produces less network traffic. The alternative for leaving nodes is to explicitly removed the data object metadata in the DHT.

## 4.5 Grid nodes

In addition to the storage layer software, a grid node has Globus Toolkit (version 4) installed or another, compatible, resource virtualization software. A GridFTP service based on GSI is running on a grid node to provide secure external access to files located on that node. Grid nodes must also include software that complies with the user requirements defined in Section 3. Below are possible solutions to the user requirements.

### **Automated staging of input and output files**

Globus Toolkit 4 via the job submission component or through logic provided directly by the AstroGrid-D job management. The file management must provide notification of transfers.

### **Monitoring of log-files**

The standard Globus Toolkit 4.0 services (e.g. GRAM-WS) already provide interactive monitoring access to `stdout/stderr` log-files of running grid jobs via file streaming. If possible, the same mechanisms will be used to monitor application-specific log-files;

otherwise a policy (i.e. by setting an environment variable) can be defined of where to put such log-files so that they are accessible from the frontend through the standard grid file management service.

### Remote partial file access

While the GridFTP protocol already provides commands to read/write parts of a remote file on a byte level for unstructured files, it also defines possible extensions for developers to implement their own commands for higher-level access to structured files (such as HDF5). The GridFTP server distributed with Globus Toolkit 4.0 has an extensible interface which allows developers to add such higher-level file access commands as server-side plugins. An earlier approach to provide partial access specifically to remote files in HDF5 format, the *GridFTP Virtual File Driver for the HDF5 library*[3], will be adopted for the AstroGrid-D grid file management, possibly as a user's grid service, or, if this is not feasible, as a standardized GridFTP service for AstroGrid-D grid nodes. Since the grid functionality is entirely encapsulated within the GridFTP server and the HDF5 library, this approach can be integrated transparently into existing post-processing applications.

## 4.6 Security model

The security model for the file management is very similar to how security is modeled in the Information Service. Essentially, this includes support for access control lists (ACLs). Each data object has an owner, which defaults to the object creator. The owner is the only user allowed to remove the data object. In order to allow other users or groups to access the data object, the owner needs to define an ACL for the data object. The ACL has two type of privileges, read-only and write. Users with read-only access are only allowed to read the contents of the data object, while a user with write access is also allowed to change the content. A user is defined as a holder of a valid AstroGrid-D X.509 certificate.

Each data object has a flag for public access, which can only be changed by the owner. Public access gives non-AstroGrid-D users read access to the data object. This is useful for sharing public data archives available in the storage layer, but can also be used by researchers for sharing data within a collaboration containing non-AstroGrid-D users.

## 4.7 Metadata

There are three type of metadata associated with the distributed file management. Data object metadata, which is internal metadata on data objects used by the storage layer. Service metadata, used to announce storage layer nodes to the AstroGrid-D Information Service. Finally, user metadata, which is any metadata related to data objects added by the user to Information Service. The rest of this section assumes that the reader has understood the basic concepts of RDF [15] and SPARQL [16].

```
@prefix job: <http://www.gac-grid.de/schema/jobs#> .
@prefix oid: <http://storage.gac-grid.de> .
@prefix user: <http://www.gac-grid.de/users/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://jobs.gac-grid.de/id/42>
  rdf:type job:Job;
  job:location <https://astrogrid.aei.mpg.de:8443/wsrp/services/GRAM>;
  job:input oid:/test/eaglenebula.fits;
  job:minCPU "1GHz";
  job:minMEM "1GB";
  job:minDISK "30GB";
  job:output <file://$SCRATCH_PATH/result.bin>;
```

Figure 4.2: A sample job description in N3.

```
@prefix file: <http://www.gac-grid.de/schema/files#> .
@prefix oid: <http://storage.gac-grid.de> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix cactus: <http://astrogrid.aei.mpg.de/cactus/syntax#> .

oid:/test/eaglenebula.fits;
  rdf:type file:DataObject;
  file:tag "cactus";
  file:tag "wavedemo";
  file:owner "Pelle Test";
  file:location "http://astrogrid.aei.mpg.de/wavedemo/input.conf";
  file:filesize "3487";
  cactus:produced_by <http://astrogrid.aei.mpg.de/cactus/simulation#342> .
```

Figure 4.3: A sample data object description in N3.

## User metadata

In the RDF information model, statements are made about a subject, i.e. a URI representing a subject. Data object identifiers, OIDs, are unique identifiers for data objects stored in the storage layer. As discussed in Section 4.2, an OID prefixed by a base URI is a good combination for usage as an RDF subject. This allows a user to make arbitrary statements about a data object. For example, define the base URI as `http://storage.gac-grid.de` and the OID `/test/eaglenebula.fits`, then the subject URI becomes:

`http://storage.gac-grid.de/test/eaglenebula.fits`. Since an RDF subject also can be an object in the RDF graph, it is possible to associate data objects with, for example, grid jobs submitted via the AstroGrid-D job management. The following two examples illustrates both alternatives with RDF data in Notation 3 [4]. The example uses a hypothetical job description format.

Figure 4.2 shows a sample of a job description that takes an input file defined with an OID. The job description vocabulary contains a predicate “input”, which is used to make statements

about the input-files to a job. The data object is referenced within the RDF graph of a job description. In Figure 4.3, the metadata for the OID is presented. This example contains an additional Cactus statement, added by the Cactus application after simulation run was finished. This highlights the flexibility in the different ways to add metadata using the RDF model and how application-specific metadata can be mixed with the file management metadata.

## System metadata

System metadata is generated for a data object through the indexing procedure described in Section 4.3. This metadata is accessed through a lookup in the DHT.

**OID** A storage layer unique name identifying a data object.

**timestamp** A date-time assigned last time the data object changed.

**type** This is a standard RDF predicate indicating of what type the subject is. The default for data objects is “DataObject”.

**location** A network accessible URI with a location of the data object. The value is always a URI. Note that the URI may be non-accessible from time-to-time due to the best-effort characteristics of the Internet. Several location statements indicates replicated data objects.

**filesize** The exact size in bytes of the data object.

**TTL** A date-time indicating when the data object expires and can be removed from the storage layer.

**owner** The owner of a data object. This can be a single user or a group of users. See section 4.6 for more information.

**ACL** Defines a list of users and groups and their access privileges for the data object.

## Service metadata

Service metadata is announced to the AstroGrid-D Information Service with the purpose of making it easier to find storage nodes depending on their capabilities. Additionally, the information can be used to find a bootstrap node for joining the storage layer. A storage layer node add service metadata to the Information Service when it joins the storage layer.

**service address** The address of the host and port where the service is running.

**storage capacity** The amount of storage the node offer to the storage layer.

**type** This is a standard RDF predicate indicating what type the subject is. This defaults to the “StorageLayerNode” for storage layer nodes.

## 5 Service Interface

The aim of the file management interface is to enable transparent access to files using standard Internet protocols. This section will not present the exact methods that will be available in the distributed file management implementation. Instead, an overview of the methods and their planned functionality is presented. Note, these methods are for the external service interface, thus, a client side library supporting the presented interface needs to be implemented for use by applications.

### 5.1 Interface description

The signature of the methods is given in the form: <return type> <method name>([<input type> <input argument name>]) throws [<exception>]. If a method does not throw an exception then throws [<exception>] is omitted. A list is a tuple within square brackets [<list>], where the notation for a tuple is a list of <datatype name> enclosed within parentheses (<datatype name>, ..., <datatype name>). The keyword optional is used to state that the method argument is optional.

### 5.2 Data types

Data type	Description
OID	An OID identifying a data object.
URI	A URI, typically representing a data object location.
FileData	The data of a file.
ACL	An access control list.
ByteRanges	A integer list with a combination of byte ranges.

Table 5.1: Datatypes

### 5.3 Exceptions

Exceptions are used to report errors back to a client. Table 5.2 defines the exceptions for the AstroGrid-D file management.

Exception name	Description
FileNotFoundException	Thrown when the file could not be accessed with the reason that it was not available.
AuthorizationFailedException	This exception is thrown when a user does not have sufficient rights for the current access.

Table 5.2: List of exceptions

## Data object management

These are the basic methods for management of data objects.

**OID post(String name, FileData data, optional ACL acl) throws AuthorizationFailedException**

**Parameters:**

name A name identifying the data object.  
 data The file data for the new data object.  
 acl An ACL defining the access rights for the new data object.

**Throws:** AuthorizationFailedException

**Returns:** OID

Method description: Creates a new data object from the given name and data. Returns the OID that represents the new data object. If the optional argument acl is specified, then the given acl is applied directly to the new data object.

**FileData get(OID oid) throws AuthorizationFailedException**

**Parameters:**

oid The OID of the data object.

**Throws:** AuthorizationFailedException

**Returns:** DataStream

Method description: Returns the file data from the data object addressed by the given OID.

**void replicate(OID oid, URI destination) throws AuthorizationFailedException**

**Parameters:**

oid The OID of a data object.

destination The destination for the copy

**Throws:** AuthorizationFailedException

**Returns:** void

Method description: Creates a replica of the object with oid at the given destination URI.

void **archive**(OID oid) **throws** AuthorizationFailedException

**Parameters:**

oid The OID of a data object.

**Throws:** AuthorizationFailedException

**Returns:** void

Method description: Creates and archived version of the given OID.

void **delete**(OID oid) **throws** AuthorizationFailedException

**Parameters:**

oid A data object identifier.

**Throws:** AuthorizationFailedException

**Returns:** void

Method description: Deletes the data object with the given OID.

[URI] **replicaList**(OID oid) **throws** AuthorizationFailedException

**Parameters:**

oid A data object identifier.

**Throws:** AuthorizationFailedException

**Returns:** [URI]

Method description: Retrieves the physical storage locations, a set of URIs, for the data file represented with the given OID.

void **register\_replica**(OID oid, URI location) **throws** AuthorizationFailedException

**Parameters:**

oid A data object identifier.

location The data location of the externally registered replica.

**Throws:** AuthorizationFailedException

**Returns:** void

Method description: Registers a new replica location for the given OID.

void **unregister\_replica**(OID oid, URI location) **throws** AuthorizationFailedException

**Parameters:**

oid A data object identifier.  
location The data location of the externally registered replica.

**Throws:** AuthorizationFailedException

**Returns:** void

Method description: Un-registers a replica location for the given OID.

**Data object security management**

ACL **getACL**(OID oid) **throws** AuthorizationFailedException

**Parameters:**

oid The OID of a data object.  
**Throws:** AuthorizationFailedException  
**Returns:** An ACL

Method description: Returns the ACL for a given OID.

void **postACL**(OID lfid, ACL acl) **throws** AuthorizationFailedException

**Parameters:**

oid The OID of a data object.  
acl An access control list.  
**Throws:** AuthorizationFailedException  
**Returns:**

Method description: Sets the ACL for the given OID.

## 6 Scenarios

In this chapter different scenarios are presented that give a brief introduction in what different ways the distributed file management can be used from both the grid and the user perspective. These scenarios cover the most important aspects of the distributed file management; insertion of new files, access of files by using OID, retrieving replicas for an OID, partial file access and interaction with the job management. It should be noted that the scenarios are hiding details, security protocols, error messages, etc. to make the scenarios shorter.

The figures show the different actors that are part of a scenario in boxes extended with lines representing time. A flow of events is a set of arrows indicating the direction of the message or request between the actors. A dashed line represents an alternative or optional flow of events, for example an error. When an alternative event occurs before the primary flow of events, the latter flow is interrupted. Arrows going back to the same actor should be interpreted as an internal method call. Additional data passed during a request is denoted under the request line. This can, for example, be security credentials, which are not explicit arguments part of the request.

### 6.1 File access using an OID

Figure 6.1 depicts the standard scenario of accessing a file stored on a storage node part of the storage layer. It is assumed that the client knows the OID at this stage. An OID can, for example, be found by querying the Information Service. The OID sent as an argument of a get request to any storage layer node. In this case, storage layer node A received the request. Unfortunately, it does not store the file. Instead, it uses the DHT to lookup the given LFID and then decides what location to return using the metadata. The client downloads the file and verifies the integrity (optional step) of the file, if the checksum is available in the metadata.

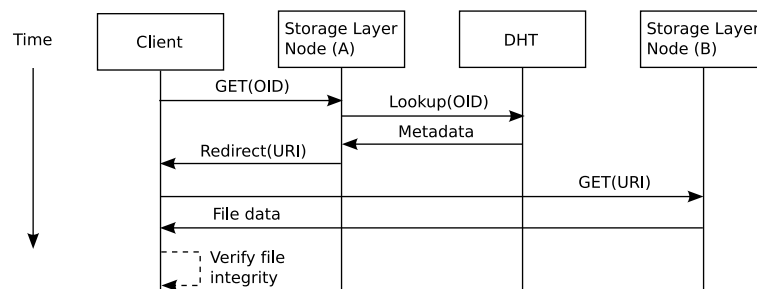


Figure 6.1: Accessing a data object using an OID.

## 6.2 Accessing a replicated data object

Figure 6.2 shows how a client can access a replicated data object stored by the distributed file management. This scenario is similar to 6.1 in the last steps. The main difference is the Replicas request to storage layer node A. This method returns a set of URIs, from which the client selects one. How an URI is selected by the client is not specified.

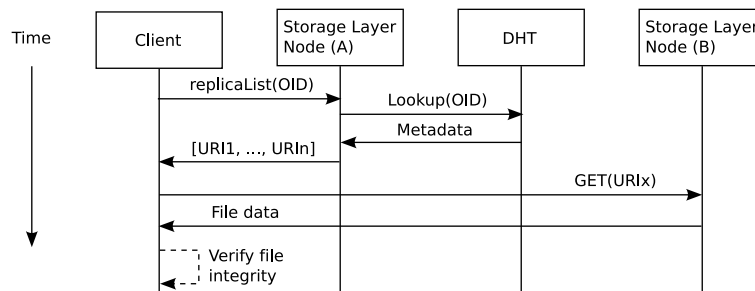


Figure 6.2: Accessing a replicated data object.

## 6.3 Upload a data object

Figure 6.3 shows the process needed for a client to upload a new file to the distributed file management. In the first step, the client posts the file to any storage layer node. The storage layer node receives the file, indexes it and generate an identifier for the DHT. Moreover, when the file is completely received, the storage layer node registers the OID and related metadata, such as location and timestamp, with the DHT. The DHT acknowledges when the (OID, metadata)-mapping is registered. In the final step, the storage layer node returns the OID to the client.

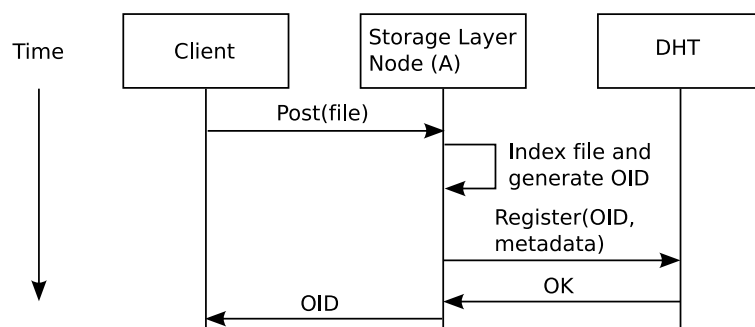


Figure 6.3: Uploading a new data object

## 6.4 Partial HDF5 file access

Figure 6.4 illustrates the steps needed for remote partial access of an HDF5-file. The client, in this case, is assumed to be a visualisation tool. The OID is known already, why the client first queries a storage layer node for the physical location of the OID. This location represented as a URI is then copy/pasted into the visualisation tool which includes a GridFTP-client. The GridFTP-client contacts the GridFTP-Server with the user's credentials for authentication and then it sends a partial HDF5 file read request to the server. The GridFTP-Server processes the request and, if successful, returns the requested HDF5 data, otherwise an error occurs (not showed in the figure) such as authorisation to access the file failed, file does not exist or the request contained invalid request parameters.

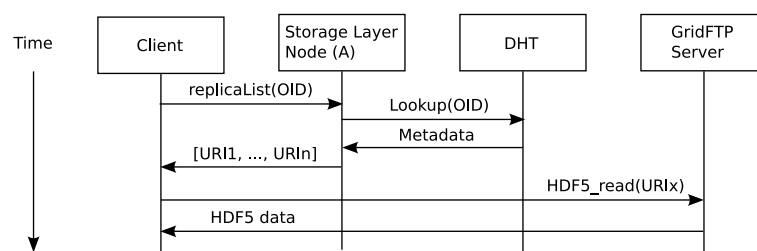


Figure 6.4: A client accesses an HDF5 file for visualisation.

## Bibliography

- [1] dCache. <http://www.dcache.org/>.
- [2] Grid Security Infrastructured (GSI). <http://www.globus.org/security/overview.html>.
- [3] Gridftpvfd-hdf5: Gridftp virtual file driver for the hdf5 library. <http://www.cactuscode.org/old/VizTools/SFTPD-HDF5.html>.
- [4] Notation 3. <http://www.w3.org/DesignIssues/Notation3>.
- [5] The SDSC Storage Resource Broker (SRB). <http://www.sdsc.edu/srb/>.
- [6] Szymon Acedanski, Supratim Deb, Muriel Medard, and Ralf Koetter. How Good is Random Linear Coding Based Distributed Networked Storage? In *NetCod '05, Proc. of the First Workshop on Network Coding, Theory and Applications*, April 2005.
- [7] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Far-site: federated, available, and reliable storage for an incompletely trusted environment. In *OSDI '02: Proc. of the 5th symposium on Operating systems design and implementation*, pages 1–14, New York, NY, USA, 2002. ACM Press.
- [8] R. Chinnici, J-J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. <http://www.w3.org/TR/wsd120/>, March 2006.
- [9] Alexandros G. Dimakis, Vinod Prabhakaran, and Kannan Ramchandran. Decentralized erasure codes for distributed networked storage. *IEEE/ACM Trans. Netw.*, 14(SI):2809–2816, 2006.
- [10] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [11] I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, and S. Weerawarana. Modeling stateful resources with web services v. 1.1. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf>, March 2004.
- [12] M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, and H. F. Nielsen. SOAP version 1.2 part 1: Messaging framework. <http://www.w3.org/TR/soap12-part1/>, June 2003.
- [13] Mikael Höggqvist and Thomas Röblitz. AstroGrid-D Information Service, Requirements Specification and Architectural Design, July 2006.

- [14] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, New York, NY, USA, 2004. ACM Press.
- [15] F. Manola and E. Miller. RDF primer. <http://www.w3.org/TR/rdf-primer/>, February 2004.
- [16] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. <http://www.w3.org/TR/rdf-sparql-query/>, February 2006.
- [17] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.
- [18] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, February 2003.