GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Fakultät für
Physik

Master's Thesis

HappyFace Meta-Monitoring für ATLAS im Worldwide LHC Computing Grid

HappyFace Meta-Monitoring for ATLAS in the Worldwide LHC Computing Grid

prepared by

Christian Georg Wehrberger

from Bloomington, Indiana (USA)

at the II. Institute of Physics
Georg-August-University Göttingen

Thesis number: II.Physik-UniGö-MSc-2013/07

Thesis period: 1st January 2013 until 13th December 2013

Supervisor: Prof. Dr. Arnulf Quadt

First referee: Prof. Dr. Arnulf Quadt

Second referee: Dr. Kevin Kröninger

Abstract

Um die bei den vier größten Experimenten ALICE, ATLAS, CMS und LHCb des Large Hadron Collider (LHC) am CERN-Forschungszentrum anfallenden Daten speichern und analysieren zu können, wurde das Worldwide LHC Computing Grid (WLCG) eingerichtet. Der stabile Betrieb von Grid-Rechenzentren im WLCG spielt eine zentrale Rolle für die beteiligten Experimente als grundlegende Infrastruktur. Bedingt durch Komplexität und Ausdehnung stellt ihr Betrieb eine große Herausforderung dar und erfordert Administration und durchgängige Überwachung aller Hardware, Software und Grid-Dienste, die für den Betrieb notwendig sind. Dazu wird von einer Vielzahl von Überwachungssoftware Gebrauch gemacht. Das modulare Meta-Überwachungswerkzeug HappyFace bietet einen einzigen Zugangspunkt zu gebündelten und bewerteten Überwachungsdaten. Diese Masterarbeit beschäftigt sich mit der Einrichtung von HappyFace Version 3 im ATLAS Tier-2-Zentrum GoeGrid an der Georg-August-Universität Göttingen und der Entwicklung von HappyFace-Modulen zur Überwachung von ATLAS WLCG Rechenzentren. Weiterhin behandelt diese Abschlussarbeit die Definition und Implementation zweier Webdienste auf Basis von WSDL/SOAP bzw. REST, welche die von einer HappyFace-Instanz gesammelten und gespeicherten Daten verfügbar machen.

Stichwörter: Physik, WLCG, ATLAS, Grid Computing, GoeGrid, HappyFace, Meta-Monitoring, Webdienst, WSDL, RESTful

In order to store and analyse all data gathered by the four main experiments ALICE, ATLAS, CMS, and LHCb at the Large Hadron Collider (LHC) at CERN, a distributed computing infrastructure, the Worldwide LHC Computing Grid (WLCG), was established. The stable operation of grid computing centres in the WLCG plays a key role in all experiments as a fundamental infrastructure. Due to their complexity and extent, their operation represents a major challenge and requires administration and continuous monitoring off all hardware, software, and grid services run. For this purpose, a large variety of monitoring tools is used. The modular meta-monitoring framework HappyFace provides a single point of access to summarised and assessed monitoring data. This Master's Thesis deals with the setup of HappyFace version 3 in the ATLAS Tier-2 Centre GoeGrid at the Georg-August-University Göttingen and the development of HappyFace modules for ATLAS WLCG resource centres. In addition to that, this thesis defines and describes key aspects in the implementation of a WSDL/SOAP-based and a RESTful web service for access to all data acquired and stored by a HappyFace instance.

Keywords: Physics, WLCG, ATLAS, Grid Computing, GoeGrid, HappyFace, Meta-Monitoring, Web Service, WSDL, RESTful

Contents

List of Figures

List of Tables

1. Introduction

The *Worldwide LHC Computing Grid (WLCG)* [3] provides a computing infrastructure for the four main experiments *ALICE* [4], *ATLAS* [5], *CMS* [6], and *LHCb* [7] of the *Large Hadron Collider (LHC)* [8], a proton-proton collider with a design centre-of-mass energy of 14 TeV. As part of the WLCG, more than 150 computing centres are distributed in 35 countries. For the preservation and processing of huge amounts of experimental data, computing resources and mass-storage are provided by each of these sites. The WLCG is an open system for the more than $8,000$ physicists involved in the LHC collaborations. Regardless of their physical location, all collaboration members have access to the distributed computing power.

The *ATLAS Distributed Computing (ADC)* infrastructure is part of the WLCG, organised in a so-called *Tier* [9] structure. Several central key services guarantee its functionality, most important the *ATLAS Production and Distributed Analysis System (PanDA)* [10] for job submission and distribution to the ADC sites. Alongside the production system, the *Distributed Data Management (DDM)* [11] system is used. Both, the WLCG and the ADC, have proven to be stable and robust systems. Since its active use for experimental data taken at the LHC from 2008, these infrastructures have never experienced a general failure or downtime. In the Tier-0 centre at CERN, the PanDA system and the central DDM service are hosted. The Tier-0 centre is the head of the WLCG Tier structure and hence stores raw experimental data, performs data calibration, and first data processing steps. From the Tier-0 centre, data are further distributed to the subordinated Tier-1 sites. These computing centres form the second layer of the WLCG infrastructure. In ADC Tier-1 centres, raw experimental and simulated data are stored. Furthermore, data are reprocessed and bulk analysis jobs are executed. A share of the data stored in Tier-1 centres is passed to Tier-2 centres. These are dedicated to perform a number of central tasks, such as Monte-Carlo simulation, analysis, and calibration of data. Apart from their design tasks, Tier-2 centres also process on-demand user simulation and analysis jobs. The lowest, though unofficial layer of the Tier infrastructure is constituted by the Tier-3 centres. For these computing centres, there is no *Memorandum of Understanding,* as for Tier-1 and Tier-2 sites, which specifies its contribution to the WLCG and its duties formally. Most commonly, user analysis tasks are performed and user data are stored.

High throughput and the stable performance of ADC in terms of the WLCG are the result of efficient management, monitoring of the entire infrastructure, and proper action taking on each level of the Tier structure. In this context, the concepts of monitoring and meta-monitoring, as well as the meta-monitoring framework *HappyFace* [12] will be described in the course of the next chapter, followed by a description of the WLCG ATLAS Tier-2 centre *GoeGrid* [13] at the *Georg-August-University Göttingen.* The use of the meta-monitoring tool HappyFace at GoeGrid and the development of generic HappyFace modules for ADC grid sites are the main foci of this thesis and therefore are covered in a separate chapter.

The world wide web is designed for the interaction of applications with humans. For the interaction of applications with each other via the internet, web services are used. Two different concepts are widely used: lightweight, intuitively usable RESTful web services and highly standardised WSDL/SOAP-based web services, which are a recognised standard in the WLCG community. These concepts were made use of for the development of web services for access to the data stored by HappyFace in order to make it available for further analysis.

2. GoeGrid - A WLCG ATLAS Tier-2 Centre

The departments and institutes of the Georg-August-University Göttingen represent various scientific communities. The demand from these communities on a grid computing infrastructure lead to the joint project GoeGrid, a computing facility sharing its resources amongst its contributors. GoeGrid is hosted at the *Gesellschaft für Wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG)* [14] computing centre, which supplies GoeGrid with electrical power, a central cooling infrastructure, and a high-bandwidth network connection. Currently, the main consumers of GoeGrid computing resources are the high energy physics community [15] and the department of physics[1]. Besides providing computing resources to all above mentioned communities, GoeGrid is a WLCG Tier-2 centre and part of the ADC infrastructure. In this chapter, detailed information on the GoeGrid hardware, software setup, and the monitoring tools used and developed at GoeGrid will be given. The current apportionment of GoeGrid computing resources is shown in figure 2.1.

[1]Institute of Theoretical Physics, Georg-August-University Göttingen

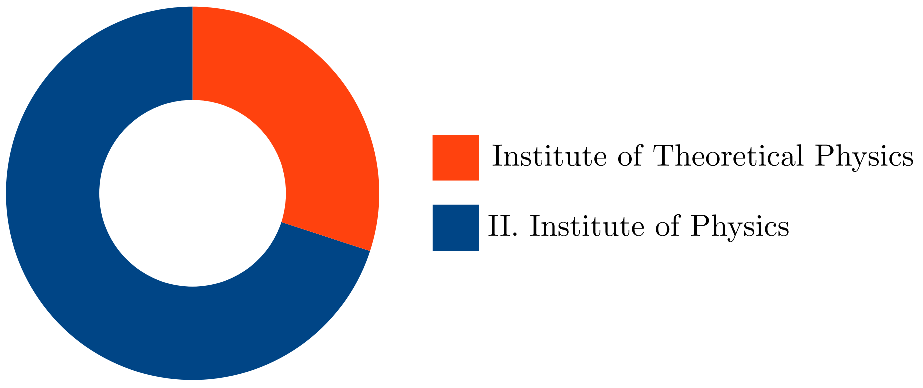


Figure 2.1.: Design resource share of the GoeGrid resource centre.

CPU	**Clock [GHz]**	**#CPUs**	**#Cores**
2 *Intel Xeon X5355*	2.66	105	840
2×2 *Intel Xeon E5440*	2.83	95	760
2×2 *Intel Xeon E5530*	2.4	16	128
2×2 *Intel Xeon X5650*	2.66	56	672
2×2 *Intel Xeon E5-2660*	2.2	16	256
2×2 *Intel Xeon E5-2665*	2.4	16	256

Table 2.1.: Current GoeGrid CPU computing resources.

2.1. Hardware Setup at GoeGrid

A typical WLCG resource centre provides resources for computing and data storage. The GoeGrid computing cluster hosts 2912 CPU-cores for all communities and about 1.1 PBytes of storage space for the ATLAS community only. The CPU-cores, each granted a share of $3 - 4$ GBytes of memory, are distributed among 305 worker nodes. Table 2.1 displays the CPU type and quantity of CPUs used for GoeGrid. 15 heterogeneous servers manage the storage space, to each of which six *RAID 6* hard drive pools are attached.

The compute nodes and the storage servers are interconnected via a 10 Gb/s network. Only the storage servers have a fully qualified domain name and an internet connection. Through a network bonding device, two of the network cards per server offer a failover against the failure of one of them.

2.2. Software Setup at GoeGrid

For the operation of a WLCG ATLAS Tier-2 centre, a variety of software and services are necessary. In this section, the GoeGrid software setup will be described, with an emphasis on the part serving as a WLCG ATLAS Tier-2 centre.

GoeGrid is running a *Red Hat Enterprise Linux* [16] distribution as an operating system for all its hardware: all hosts run the 64-bit *Scientific Linux CERN (SLC)* [17] version 6.4. For easy maintenance of routine tasks, as well as the maintenance and expansion of the GoeGrid infrastructure, there is a number of services and software in use. In terms of routine task management, like certificate distribution, firewall status checks, or spreading changes in configuration files, *CFEngine*[2] is used. It is installed on all GoeGrid hosts and

[2] An open-source configuration management tool [18].

guarantees the robustness of the system. In conjunction with CFEngine, *Rocks*[3] is used for the registration of new hosts in the *DNS*[4] and *DHCP*[5] servers. Along with hardware servers, the virtualisation technique *KVM*[6] is actively used at GoeGrid. Currently, four hardware servers are combined to a single virtualisation cluster, which hosts 11 virtual servers.

All above mentioned systems heavily use *NFS*[7] and *Autofs*[8]. In combination with NFS, Autofs is used to distribute certificates or configuration files by CFEngine. For the virtualisation cluster, NFS provides a central repository for all virtual machine disk images.

2.3. Grid Services at GoeGrid

GoeGrid is part of the WLCG structure as an ATLAS Tier-2 and Tier-3 computing centre. Thus, along with the above mentioned systems, it is running all required WLCG and ADC services.

According to the WLCG central services, the GoeGrid key services can be identified. In terms of data management, a Tier-2 centre provides interfaces for file transfers via different services, such as *dCap*[9], *GridFTP*[10], *GSIFTP*[11], *GSIdCap*[12], and *SRM*[13]. For the data management at GoeGrid, the system *dCache* [27] is used, also widely used in the German ADC cloud. The system dCache provides all protocols required by the central *Distributed Data Management (DDM)* [11] system and the advantage to unify heterogeneous storage units to one centrally accessible storage system. As in many ADC Tier-2 centres, dCache is configured to split the GoeGrid storage space in several space tokens: *PRODDISK*, *SCRATCHDISK*, *LOCALGROUPDISK*, and *DATADISK*, which are under DDM management.

[3] An open-source cluster distribution system [19].
[4] *Domain Name System*
[5] *Dynamic Host Configuration Protocol*
[6] *Kernel-based Virtual Machine* [20]
[7] *Network File System*
[8] An automatic file system mounter [21].
[9] *dCache Access Protocol* [22]
[10] *Grid File Transfer Protocol* [23]
[11] *Grid Security Infrastructure* [24]
[12] *Grid Security Infrastructure dCache Access Protocol* [25]
[13] *Storage Resource Manager protocol* [26]

In order to accept and execute analysis or Monte-Carlo simulation (production) jobs, it is important to provide a front-end for the communication with the grid workload management system. In case of ADC, in general PanDA is used, possibly alongside other systems. At GoeGrid, two *Computing Elements*[14] constitute a grid jobs front-end to the local batch system. These servers are also responsible for local user mapping and the publishing of accounting information to the local *APEL*[15] server. As a batch system, *Terascale Open-Source Resource and QUEue Manager (TORQUE)* [31] is used. For more advanced job scheduling, it is interfaced with the *Maui* [32] cluster scheduler. As the GoeGrid computing centre resources are shared by several communities, a *fair* way for the utilisation of these resources needed to be found. The Maui *fairshare* algorithm is configured in a way that the time-integrated usage of GoeGrid resources should comply with the level of investment and financial aid that the different user groups provide.

In addition to the services mentioned above, GoeGrid also hosts the *BDII*[16] and the APEL server, designed for site configuration and the publishing of performance information.

2.4. Monitoring at GoeGrid

Due to the extents of its infrastructure and the large number of systems and services, the stable and efficient operation of GoeGrid is a major challenge. In order to address this challenge, it is highly important to detect the failure of hardware or critical services in time and identify its root cause. These are tasks of monitoring systems.

On the one hand, different monitoring tools are in use for the detection of hardware or local service failures, e. g. *Nagios* [34], or the detection of a system degradation, e. g. *Ganglia* [35]. On the other hand, the ADC services with respect to GoeGrid require monitoring via their *GUI*[17]. Besides that, the meta-monitoring software HappyFace is in active use at GoeGrid. Providing a single point of access to various monitoring sources via specific modules, it makes all critical information easily accessible. The module and category rating schema allow for the detection of failures at first glance and the correlation of problems. HappyFace monitors the GoeGrid hardware and site-related grid services,

[14] *EMI (European Middleware Initiative)* [28] *CREAMCE (Computing Resource Execution And Management Computing Element)* [29]
[15] *Accounting Processor for Event Logs* [30]
[16] *Berkeley Database Information Index* [33]
[17] *Graphical User Interface*

which makes it a key component in cluster management and maintenance.

3. The HappyFace Meta-Monitoring Project

3.1. Monitoring

The act of performing computer surveillance of computing resources is called *monitoring*. The analysis of monitoring data is responsible for the detection, comprehension, and rectification of failures related to computing resources. This includes computing hardware, computing infrastructures, software, and services. By monitoring, failures can be detected before they become problems. Consequently, their root cause is identified and rectified. Besides that, monitoring also serves for measuring the performance of computing resources, using adequate metrics. For the description of monitoring (in grid environments), the *Global Grid Forum (GGF)*[1] defines a number of terms introduced in the following.

3.1.1. The Grid

The concept of grid computing was initially designed by Ian Foster and Carl Kesselman in 1998 [37]. They proposed a new computing infrastructure, designed for the coordination of distributed resources with regard to applications in astronomy, biology, medicine, engineering, and high-energy physics. In order to distinguish the grid from other, related terms, it is defined as follows [37]:

- *Coordination of distributed resources*: A grid integrates and coordinates distributed resources. In that respect, it takes care of security, policies, memberships, and other related issues.
- *Use of standard, open, general-purpose protocols and interfaces*: A grid is established by the use of various protocols and interfaces. They address e. g. authentication,

[1]The GGF was merged with the *Enterprise Grid Alliance* in 2006 and is now the *Open Grid Forum (OGF)* [36].

authorisation, resource discovery, and access.

- *Delivery of non-trivial qualities of service*: In coordination, the grid resources offer various qualities of service. In that respect, the combined resources of a grid are designed to be more useful than the sum of its individual parts.

A grid makes use of *middleware* in order to accomplish this. Middleware is a software, which enables the application-neutral communication and data management. Therefore, it hides the grid complexity by constituting a high-level service provider to arrange the communication between all grid software and hardware components.

3.1.2. Terminology

The accurate description of monitoring in grid environments requires the definition of terms and concepts. An *entity* is defined as a unique component integrated into the computing infrastructure, such as processors, computer memory, storage mediums, network links, applications, and processes. *Events* define a set of timestamped data with a specific structure and are associated with entities. By *sensors*, events are generated in terms of the monitoring of an entity. There are active sensors, conducting measurements themselves, and passive sensors, mostly providing information from operating systems. With each event, also an *event type* is associated, which uniquely maps to an event structure; an *event schema* defines the structure and semantics of events.

3.1.3. The Monitoring Process

In distributed systems, such as grids, the process of monitoring consists of four phases [38]:

1. **Generation of events**: Acquired from entities, monitoring data are encoded according to predefined schemata.

2. **Processing of generated events**: Data are filtered and assessed using appropriate metrics. This step can take place at any stage of the monitoring process.

3. **Distribution**: Processed data are provided to interested parties.

4. **Presentation/consumption**: Acquired data are presented, e. g. via a GUI or in a machine-readable format.

3.1.4. Monitoring Requirements

A number of general requirements is important for every monitoring system [38]. However, their concrete realisation strongly depends on the individual use case.

- **Scalability**: A monitoring system should be configurable and adjustable to any growth of a computing infrastructure. In that respect, the response time of a scalable monitoring system should not increase significantly and reduce its performance, even though the expansion of an infrastructure induces additional load on the monitoring system.
- **Extensibility**: A monitoring system should be adjustable to new event schemata. If changes are applied to an infrastructure or parts of it, it should be possible to change the event preprocessing and presentation formats in the monitoring system configuration.
- **Data-delivery models**: A monitoring system provides a constant stream of data, that was aggregated from entities via *push* or *pull* models, which should both be supported. In push models, the data transfer request is initiated by the publisher of the data, in pull models, the receiver actively queries the data publisher.
- **Portability**: In particular, a monitoring system's sensors should be able to aggregate events independently from the environment or platform used.
- **Security**: Security services, such as access control, unilateral or mutual authentication, are scenarios of event collection, which need to be supported by a monitoring system. In that context, time synchronisation is highly important (consistency of data, expiration).

3.1.5. The Grid Monitoring Architecture

This section will provide an overview of the *Grid Monitoring Architecture (GMA)* [39], as proposed by the GGF. Its aim is to describe monitoring architectures in grid environments on the basis of common definitions. The GMA defines the roles in the process of monitoring and describes their interactions. There are five roles: *producer*, *consumer*, *registry*, *republisher*, and *schema repository.* The according actions between instances of producer and consumer roles are *publish/subscribe*, *query/response*, and *notify.*

A producer constitutes a process that implements at least one *Application Programming Interface (API)* in order to provide events. A consumer receives these events, providing

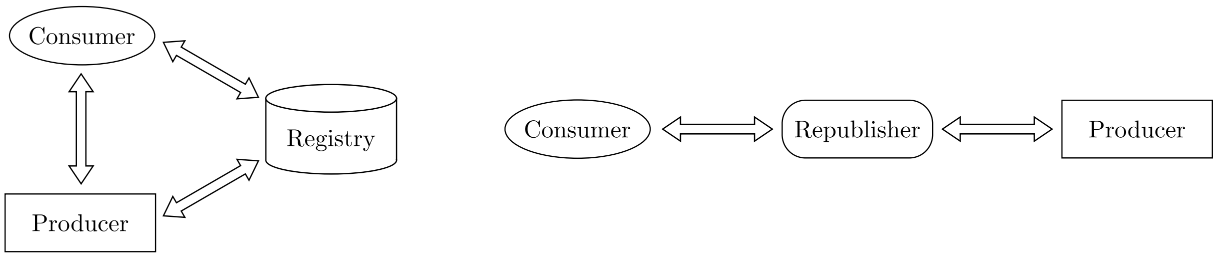


Figure 3.1.: The Grid Monitoring Architecture.

a consumer API. Producers can register event types at the registry, which consumers query for obtaining information on available event types. The registry furthermore provides details regarding the communication, such as addresses, protocols, and security requirements. Republishers are defined as components that feature producer and consumer interfaces. They may filter, aggregate, summarise, broadcast, and cache events. A schema repository is designed to hold event schemata for the collection of defined event types. In figure 3.1, the GMA components and their connections are shown.

The registry enables producers and consumers to establish a direct communication. Both producers and consumers provide their event types and event types interests, respectively, to the registry. In the same vein, all communication details are stored at the registry. Once producers and consumers have discovered each other, they communicate directly, using the information on addresses, protocols, and security requirements they have previously provided to the registry. The action publish/subscribe consists of three phases, involving one producer and one consumer. Both involved parties can initiate this action. Firstly, the consumer subscribes for a specific event type. Subsequently, the producer generates a stream of events. Lastly, the subscription is terminated, either by the consumer or the producer. While the action publish/subscribe is an action with indefinite duration, the action query/response is a singular procedure initiated by the consumer. It queries the producer of events and receives a response, which contains events for specified event types. The action notify is a one-off interaction, which consists only of a single notification sent from a producer to a consumer, or vice versa [38].

3.1.6. Categories of Monitoring Systems

Previously, the phases of monitoring and the components of monitoring systems were defined and described. These GMA components can be mapped to the four phases of monitoring: generation of events, processing, distribution, and presentation/consumption. Sensors generate events and may pre-process the acquired data already. A producer

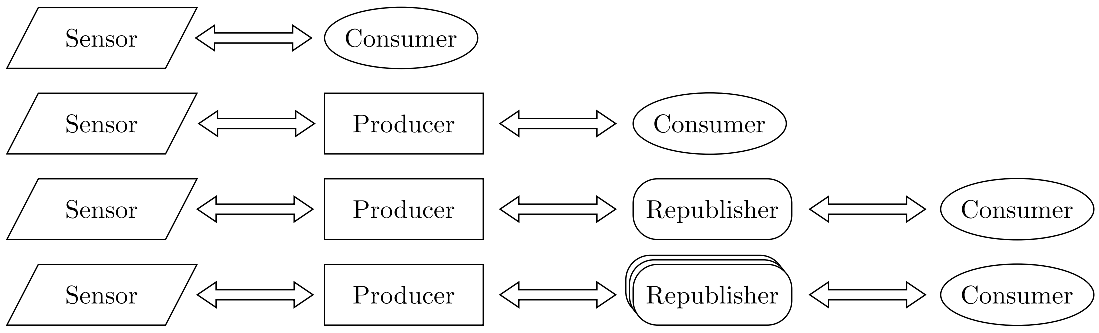


Figure 3.2.: Categories of monitoring systems: self-contained systems (level 0), producer-only systems (level 1), producer and republisher systems (level 2), and hierarchy of republishers system (level 3). The stack of republisher symbols represents a hierarchy of republishers.

may implement its own sensors and therefore generate events and process them. Nevertheless, its main purpose is the distribution of monitoring data. Republishers process events, e. g. in terms of filtering and summary, and distribute them further. A chain of at least two republishers constitutes a *hierarchy of republishers*. Besides a possible processing, a consumer performs the presentation/consumption of events.

Based on this mapping, is it possible to categorise monitoring systems. According to [38], there are four levels of monitoring systems, as shown in figure 3.2. The lowest level (level 0) of monitoring systems consists of a sensor and a consumer only. Events are passed from sensor to consumer either *on-line*, i. e. at the same time the events are generated, or *off-line*, i. e. afterwards. On this level, typically no generic API is involved and the information is presented via a web interface. Monitoring systems on this level are so-called *self-contained systems*. Level 1 monitoring systems, *producer-only systems*, contain a producer between a sensor and a consumer. In general, this consumer provides an API. In *producer and republisher systems* (level 2), the producer publishes its information to a republisher, which may undertake certain processing steps. Level 3 monitoring systems (*hierarchy of republishers systems*) involve an arbitrary number of republishers arranged hierarchically.

3.2. Basic Principles of Meta-Monitoring

For the monitoring of grid architectures, various monitoring systems of different levels of the GMA are available. They allow the monitoring of local hardware, local software and services, and the local infrastructure of a grid centre. In the WLCG, all involved grid centres are part of a high level Tier structure. On all levels of this structure, services provide information about single grid centres, such as GoeGrid. This variety of monitoring sources generates a large number of data streams, which need to be analysed. These streams are neither centrally accessible, nor are they accessible via similar producer APIs. Therefore, it is one of the main goals of meta-monitoring to aggregate monitoring data, and thereby reduce the amount of time spent by administrators for the checking of monitoring sources. The previously stated general requirements for monitoring systems are extended by certain properties that a meta-monitoring system preferably fulfils [12]:

- **Single point of access**: All relevant monitoring information is centrally accessible, e. g. via a single web interface.
- **Up-to-date monitoring information**: Monitoring information is preferably provided in real-time.
- **History functionality**: A history functionality gives users the ability to review monitoring results from the past and correlate it with recent information.
- **Fast accessibility**: Quick access to the monitoring information is required without much overhead.
- **Comfortable usage**: Users should be provided easy access to the monitoring information.
- **Simple warning system**: A warning system should notify the responsible persons and display status information simply and unmistakeably.
- **Modular structure**: A modular structure allows to include any type of monitoring data from various levels of the computing infrastructure.

3.3. The HappyFace Meta-Monitoring Project

HappyFace is a meta-monitoring tool that satisfies the requirements precedingly stated for monitoring tools in general and meta-monitoring tools specifically. The HappyFace project is a joint project of Karlsruhe Institute of Technology (KIT) and Georg-August-University Göttingen. In terms of monitoring categories as defined by the GMA, HappyFace is a producer and republisher system (level 2), as its description will prove in the following.

The most recent version of HappyFace is HappyFace version 3, which was released in early 2013 to replace its predecessor HappyFace version 2. The current core development is a joint task of KIT and Georg-August-University Göttingen. The development of new modules takes places at KIT, Georg-August-University Göttingen, and the University of Aachen.

HappyFace is a meta-monitoring tool designed for the aggregation, processing, and storing of monitoring data from arbitrary data sources. Thus, HappyFace is structured in separate modules, which are embedded in a core framework. Each module is responsible for the aggregation and processing of data from specific monitoring sources. The HappyFace web page publishes the module output and provides a single point of access to the monitoring information. The HappyFace core framework, the individual modules, and the HappyFace web page are individually customisable to site-specific requirements. In the following sections, the HappyFace (version 3) workflow, web page, configuration, and, in more detail, the HappyFace module development will be described.

3.3.1. HappyFace Version 3

Different from its predecessor HappyFace version 2, the HappyFace version 3 framework is written in *Python*[2] only. The use of Python with embedded *PHP*[3] in HappyFace version 2 constituted several disadvantages, as the inconvenient and error-prone development of the core framework and its modules. Besides that, HappyFace version 2 exhibited structural shortcomings. Due to a directory structure that mixed up code and configuration directories, the versioning management turned out to be a difficult task. Furthermore, several functionalities, e. g. certificate authorisation, were not fully integrated in the HappyFace core, since they were implemented later. The new version of HappyFace, HappyFace version 3, addresses these issues and is written consistently in Python. It clearly separates configuration and source code files. In addition to that, it provides an automatically generated documentation.

The HappyFace Workflow

The HappyFace workflow describes the chronological sequence of actions in the HappyFace framework. In succession, two Python scripts need to be executed. The first script `acquire.py` reads all HappyFace configuration files and executes the individual modules, which aggregate, process, and store monitoring data. Subsequently, the script `render.py` accesses the stored data and calls a specific function of each module in order to generate an

[2]An interactive, object-oriented, and extensible programming language [40].

[3]*Hypertext Preprocessor* (a recursive acronym abbreviated PHP) is a free server scripting language for generating dynamic and interactive web pages.

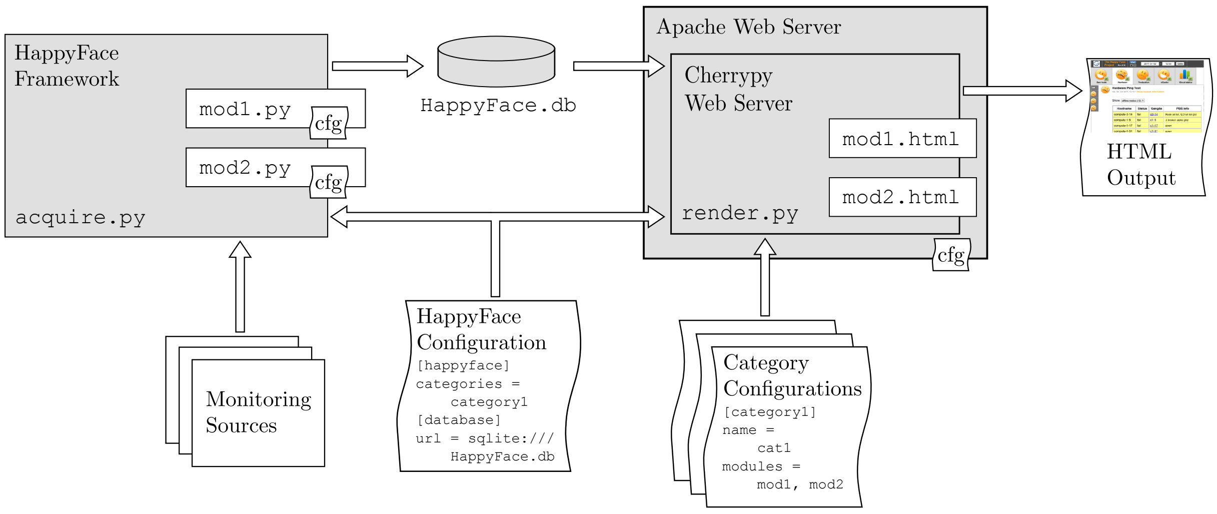


Figure 3.3.: Schematic workflow of HappyFace version 3.

output for the HappyFace web page. `render.py` also takes care of rendering the web page and makes it accessible via a web server. The HappyFace workflow is depicted in figure 3.3.

Typically, the Python script `acquire.py` is periodically executed every 15 minutes, e. g. via a *cron job*[4] on Linux/Unix operating systems. When a higher timeliness of data is required, a shorter time interval needs to be chosen. Initially, all configuration files are read in. Locally defined configuration files are given preference to default configuration files. Then, each module Python source code is executed. In separate sections of the module source code, the module is initialised, provided its configuration parameters, downloads are prepared, monitoring data are extracted, and the time-stamped data are stored to the HappyFace database[5] `HappyFace.db`.

The `render.py` Python script is responsible for the generation of a human-readable output and its presentation. Each module implements a function that accesses the previously stored data in the HappyFace database. The render script executes this function of each module and provides the extracted data to the module *HTML*[6] template file. The generated contents are then inserted into the HappyFace web page skeleton. Furthermore, `render.py` starts a local *cherrypy*[7] web server. This can be integrated into other web servers, such as the commonly used *Apache*[8] web server.

[4]A time-based job scheduler in Linux/Unix operating systems.
[5]HappyFace uses *SQLite* [41] by default.
[6]*HypertText Markup Language* [42], the most commonly used markup language for creating web pages.
[7]A Python-based, object-oriented web framework [43].
[8]An open-source HTTP server [44].

The HappyFace Web Page

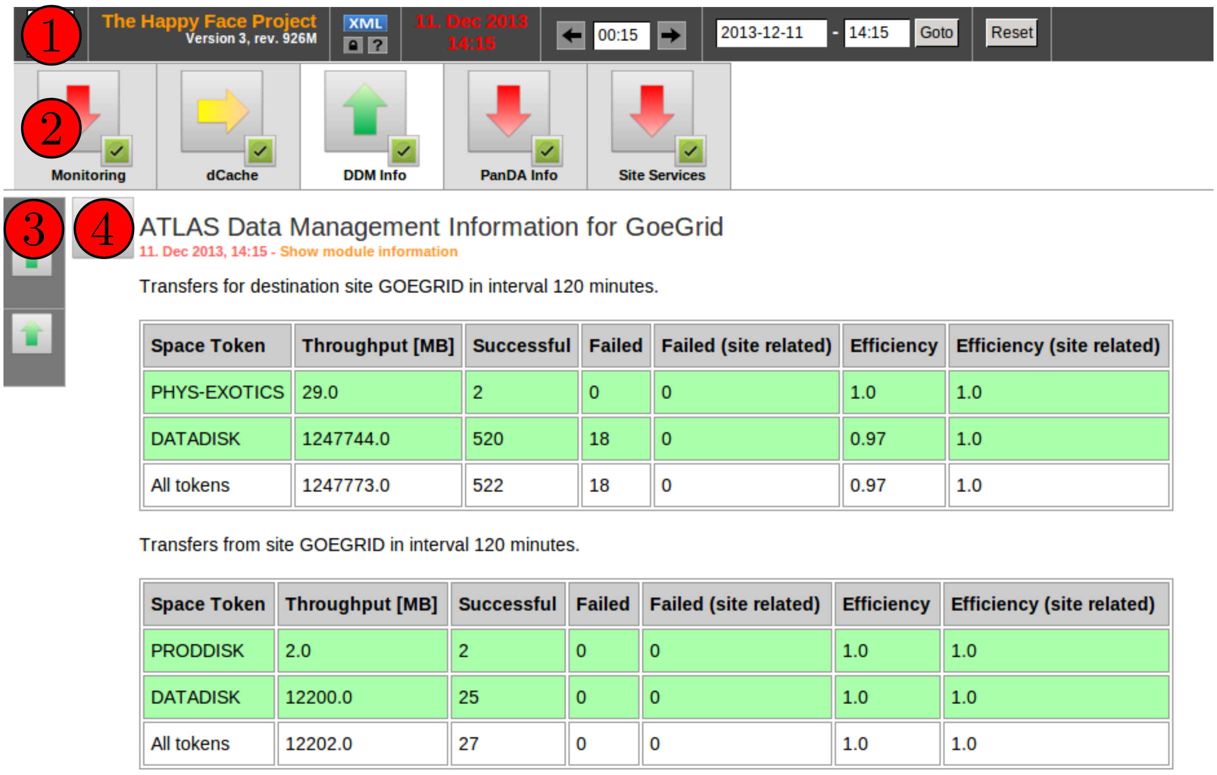
The HappyFace web page consists of a title bar and history navigation, a category navigation bar, a fast navigation bar, and the content of the individual modules. Categories are loaded on separate pages, and module details are loaded only on demand with *AJAX*[9]. This allows for the fast accessibility of the HappyFace web page, especially on mobile devices.

The history navigation bar is designed for accessing monitoring information that was stored during previous data acquisition phases in the HappyFace database. The proposed interval of 15 minutes for running `acquire.py` is also the default value for the time interval in the history navigation. By the use of certificate authorisation, the access to entire categories can be secured. In the category configuration file, the access to a category can be set to *open*, *permod*, or *restricted*. The default value *open* allows full access to any user. If *permod* is specified, the category can be accessed in general, but the access to particular modules may be restricted. In case the option *restricted* is chosen, only authorised users are granted access to the entire category. Mainly, the categories contain the individual HTML output of modules. For fast access to specific modules and an overview of the status of all modules, a fast navigation bar is integrated into each category. In addition to that, all modules provide further information, e. g. on their float status value and data sources.

HappyFace Prerequisites

Primarily, HappyFace is designed for the use on Linux/Unix operating systems. It was tested for the deployment on the Linux operating systems Ubuntu 12.10 and CentOS 6.3. HappyFace is developed in Python 2.6. Besides the correct Python version, it also requires the cherrypy 3 Python web framework. For HMTL templates, the Python template library *mako* [46] is necessary. In addition to that, HappyFace depends on the Python *SQL*[10] toolkit *sqlalchemy*[11] and a database library, such as SQLite[12] or *PostgreSQL*[13].

[9] *Asynchronous JavaScript and XML* [45]
[10] *Structured Query Language*
[11] A Python SQL toolkit and object relational mapper [47].
[12] A software library implementing an SQL database.
[13] An object-relational database management system [48].



ATLAS Data Management Information for GoeGrid

11. Dec 2013, 14:15 - Show module information

Transfers for destination site GOEGRID in interval 120 minutes.

Space Token	Throughput [MB]	Successful	Failed	Failed (site related)	Efficiency	Efficiency (site related)
PHYS-EXOTICS	29.0	2	0	0	1.0	1.0
DATADISK	1247744.0	520	18	0	0.97	1.0
All tokens	1247773.0	522	18	0	0.97	1.0

Transfers from site GOEGRID in interval 120 minutes.

Space Token	Throughput [MB]	Successful	Failed	Failed (site related)	Efficiency	Efficiency (site related)
PRODDISK	2.0	2	0	0	1.0	1.0
DATADISK	12200.0	25	0	0	1.0	1.0
All tokens	12202.0	27	0	0	1.0	1.0

Figure 3.4.: HappyFace version 3 instance for the monitoring of the Tier-2 centre GoeGrid: title bar and history navigation (1), category navigation bar (2), fast navigation bar (3), and individual module content (4).

HappyFace Installation

In the WLCG Tier-2 centre GoeGrid, HappyFace version 3 is installed on CentOS 6.3 (64-bit), an operating system widely used in high energy physics communities. For the fulfilment of the aforementioned prerequisites, the following software packages need to be installed: `python-cherrypy3`, `python-sqlalchemy`, `python-migrate`, and `python-mako`. When the use of the HappyFace plot generator is considered, the packages `python-numpy` and `python-matplotlib` require installation. The possible integration into the Apache web server makes use of the web server module `mod-wsgi`[14]. Finally, the HappyFace source code can be downloaded from a *subversion*[15] repository hosted by KIT or an *RPM*[16] package provided by the Georg-August-University Göttingen can be used. Similarly, a set of modules can be installed. The easy installation of preconfigured modules for the ATLAS community is facilitated by an according RPM package.

[14] A *Python Web Server Gateway Interface (WSGI)* adapter module for Apache [49].
[15] An open source version control and revision system [50].
[16] *Red Hat Package Manager* [51]

Further installation details are provided in the appendix, see appendix A.1.

HappyFace Configuration

The HappyFace configuration takes place in separate configuration directories. First of all, the `defaultconfig` directory is accessed by the HappyFace framework, evaluating all files with a `.cfg` suffix in alphabetical order. Whenever configuration parameters appear twice in different files, the alphabetically latter configuration file is given preference.

When HappyFace is newly set up, there is a default core configuration file `defaultconfig/happyface.cfg`, containing all possible configuration sections and options that HappyFace can handle. The section `paths` contains configuration parameters on paths for various HappyFace directories, which can be customised. Most important is the configuration key `happyface_url`, which has to be changed from the default value `/` to the path, through which HappyFace is available on the Apache web server, if any changes were applied to the default Apache configuration. In the `happyface` section of the core configuration file, the order of displayed categories and the time interval for the safe-keeping of temporary files can be altered; also the automatic reload interval of the HappyFace web site can be changed. The access to certain kind of critical infrastructure information is preferably restricted to authorised users only. That is why in the section `auth`, a file containing authorised *distinguished names* and an authentication script can be specified. The parameters in the section `template` allow for the customisation of the displayed logo, the documentation, and the title of the HappyFace web page. In additional sections, the HappyFace database, the download service, the plot generator, and the cherrypy web server are configured. An example configuration for the HappyFace core can be found in the appendix of this document, see appendix A.2.

The default configuration for the HappyFace core is separated from site-specific configuration files. While the default configuration takes place in the directory `defaultconfig/`, there is a `config/` directory, which is designed to contain all site-specific configuration files. In both directories, all files with a `.cfg` suffix are evaluated in alphabetical order, first those from the default configuration directory. Configuration parameters specified in files read in later will overwrite previously set values. The default configuration directory is under subversion version control. When updating HappyFace, changes in this configuration will be overwritten. Since configuration files may also contain sensitive information, e. g. user names and password, the `config/` directory is excluded from version control. Thus, it is not uploaded when a certain grid site wants to share its developments via subversion.

The HappyFace categories are configured independently from the HappyFace core. By default, this takes place in `config/categories-enabled/`. Each category possesses a dedicated file. When setting the category order in the core configuration, the according category identifier has to match the section name in the category configuration file. Besides the category name and description shown on the HappyFace web page, the category rating and a rating algorithm are defined. Lastly, the modules to appear in this section are listed in the order of appearance.

Configuration files for individual module instances are stored in the `config/modules-enabled/` directory. The same module can have several instances with different configuration files, allowing for the usability of a module for different monitoring sources or scenarios. The name of the section in a module instance configuration needs to be the same as in the category configuration file. There is a number of mandatory configuration keys, such as the name of the module Python source code file, as well as a display name and description. Additionally, instructions for the use of the module, a rating, and a rating weight can be specified. Other configuration keys are determined by the module Python source code: All configuration keys that are accessed need to be specified in the configuration file, or an exception is risen and the module code execution is cancelled.

Besides the HappyFace core, module, and category configuration, the operating system environment and the Apache web server require configuration, in order to make the HappyFace web page available. For security reasons, the user executing HappyFace should not have super user privileges. That is why a dedicated HappyFace user is created. In its `home/` directory, the local instance of HappyFace is installed. An option of mod-wsgi specifies a daemon process group, which can be executed by a specified user, in the Apache configuration. In the *VirtualHost*[17] section of the Apache configuration, the HappyFace script `render.py` is assigned to the daemon process group. An exemplary Apache configuration is given in the appendix of this document, see appendix A.3.

The HappyFace Rating Schema

In order to provide a simple warning system, HappyFace implements a rating functionality, both for modules and entire categories. During the processing of data, each module i assesses the gathered information and assigns a value contained in the set

[17] An Apache configuration directive for running more than one web site on a single server.

Rating	**Rating value**
ok	$s_i \in [0.66, 1]$
warning	$s_i \in [0.33, 0.66)$
critical	$s_i \in [0, 0.33)$
module execution failure	$s_i = -1$
data retrieval failure	$s_i = -2$

Table 3.1.: HappyFace module rating schema.

$s_i \in \{-2, -1, [0, 1]\}$. The status value $s_i = -2$ refers to a failure during data retrieval, $s_i = -1$ to an error that arose in the course of executing a module Python code. However, these different statuses do not visually differ on the HappyFace web page. Whenever the module functions properly, a float value from the interval $[0, 1]$ is assigned. If the monitoring data require no further investigation, a value from the sub-interval $[0.66, 1]$ is allotted (status *happy/ok*). In case of a failure indication, a value from the interval $[0.33, 0.66)$ is chosen (status *warning*). Provided that retrieved monitoring information suggests failures, a value from the interval $[0, 0.33)$ is assigned (status *unhappy/critical*).

In case the module is defined to be *unrated* in its configuration, the assigned status value is not considered. The overall category rating s_c determines a category status from the status of the modules it contains. In the according category configuration file, a rating algorithm is defined. A category status may be calculated by one of the following algorithms: *unrated*, *worst*, or *average*. In case a category is *unrated*, the modules statuses are not taken into account. The algorithm *worst* calculates the category status as the minimum over all module statuses.

$$s_c = \min_i (s_i)$$

Thus, it reflects the status of the module with the lowest status value, i. e. the worst rating. In case the algorithm *average* is used, the category status is set to the average value of all modules it contains.

$$s_c = \sum_i (w_i \cdot s_i) / \sum_i 1$$

In that calculation, a weight w_i defined in the module configuration file is considered for each module i.

According to the rating assigned, a pictogram reflects the current status. There are

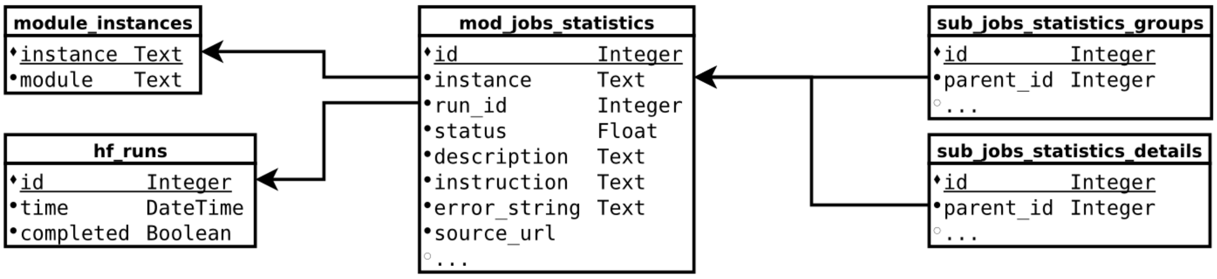


Figure 3.5.: HappyFace database schema.

several pictogram sets that can be chosen from, inter alia a set of smileys with different emotional facial expressions, from which HappyFace derives its name.

The HappyFace Database

The HappyFace database `HappyFace.db` contains two general tables, `module_instances` and `hf_runs`, not taking into account any module-specific database tables. The table `module_instances` maps instances of modules to their according module source code and HTML template file. This is important, because one module is allowed to have multiple instances, configured in different ways. Each time the Python script `acquire.py` is executed, a new table row in the table `hf_runs` is added. This row contains information on a unique *run id*, a timestamp, and a boolean flag, which indicates the successful execution or its failure.

A single module is related to at least one table, its module table. Each time the module code is executed, a new row for each instance of the module is added to the module table. It relates to a specific run via a run id which is identical with the run id entered in `hf_runs`. Thus, the module run can be uniquely identified and is associated with a timestamp. Via the module instance name table column, the module class of a specific module instance can be identified via `module_instances`. Additionally, the module table can contain information on data sources and summary information.

For storing more than single, fixed entries per run, HappyFace makes a subtable mechanism available. Each module may define an arbitrary number of subtables, containing an arbitrary number of table rows per run. Each entry in a subtable is linked to an entry in the module table. This way, each table row in a subtable can be uniquely identified and is related to an entry in `hf_runs`.

In figure 3.5, the previously described database layout is shown schematically. In the example given, only such database columns are mentioned that are common for all modules.

3.3.2. Module Development

There are four files that need to be created or edited for module development. Three of them are specific to each module, the fourth is the category configuration file. Each module needs its Python code, HTML template, and a module configuration file.

Python Source Code

This section describes how to implement a fully functional HappyFace module step-by-step. The source code of a module is a Python script. All source codes of modules are located under `/path/to/happyface/instance/modules/`. For the following paragraphs it is assumed that the name of the module to develop is *MyModule*. Consequently, the file that has to be created is `MyModule.py`. The base class skeleton defines a number of variables and functions that have to be implemented: `prepareAcquisition()` for commissioning monitoring data downloads, `extractData()` for the extraction of relevant information from the previously downloaded files, `fillSubtables()` for the insertion of extracted data into module subtables, and `getTemplateData()` for creating variables that are accessible from the HTML template.

In the Python source code, first of all the authorship and license should be defined. It is proposed to use the *Apache license*, a free software license written by the *Apache Software Foundation* [52].

```
# Copyright YEAR Institute - Institution
# Author: Your Name (your e-mail address)
#
#   Licensed under the Apache License, Version 2.0 (the "License");
#   you may not use this file except in compliance with the License.
#   You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#   Unless required by applicable law or agreed to in writing, software
#   distributed under the License is distributed on an "AS IS" BASIS,
#   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#   See the License for the specific language governing permissions and
#   limitations under the License.
```

In order to have access to HappyFace functionalities in the source code, the `hf/` directory needs to be imported, containing the HappyFace core.

```
import hf
```

For filling the related HappyFace database subtables, `sqlalchemy`[18] has to be imported.

```
from sqlalchemy import *
```

Now, the class of the module *MyModule* is defined. It inherits from `hf.module.ModuleBase`, an abstract skeleton class for all modules.

```
class ModuleName(hf.module.ModuleBase):
```

For the automatic generation of a configuration file for the developed module, the variable `config_keys` has to be defined. It is a Python dictionary with keys corresponding to the parameter names in the configuration file. Each key refers to a description and a default value.

```
    config_keys = {
            'key': ('description', 'default value')
        }
```

The variable `config_hint` is optional and defines a hint, which is added as a comment to the configuration file when it is automatically generated.

```
    config_hint = 'Give a config hint here.'
```

The columns of each module table in the database are defined as follows:

```
    table_columns = [
        Column('column_name', TEXT)
    ], []
```

While the first element of this tuple describes the database columns, the second list gives the names of columns in the corresponding module table, which point to files in the archive directory. For storing data in a more structured way, subtables can be used. When using subtables, the variable `subtable_columns` has to be defined. Each key of this dictionary corresponds to one (sub-)table in the database, named `sub_my_module_key`. As for the variable `table_columns`, the second list gives the names of columns in the according subtable, which point to files in the archive directory.

```
    subtable_columns = {
        'subtable_name': ([
        Column('column_name', TEXT)
    ], [])}
```

Each module class has to implement certain methods that are called when executing the `acquire.py` Python script. The first method to be implemented is `prepareAcquisition()`.

```
    def prepareAcquisition(self):
```

In this method, downloads of monitoring data are queued (and then performed by the HappyFace download service). In order to read a *URL*[19] from the configuration file (parameter key), the class variable `config` is accessed.

[18]A Python *Structured Query Language (SQL)* toolkit and object relational mapper [47].
[19]*Uniform Resource Locator*

```
        url = self.config['key']
```

For queuing the monitoring information download, the HappyFace download service is called, which is a HappyFace core functionality.

```
        self.source = hf.downloadService.addDownload(url)
```

The parameter `url` refers to the source of monitoring data that the module fetches information from. It needs to be of the form `local/global/both | wget-options | url`. The first part of this statement refers to the usage of local, global, or both local and global *wget*[20] parameters; in the second part, local options can be defined. The third part specifies the download source itself. It is convenient to define the list of values to be entered into subtables already at this point in the source code.

```
        self.subtable_name_db_value_list = []
```

The second method to be implemented is `extractData()`.

```
    def extractData(self):
```

In this method, data are extracted from the already downloaded files, parsed, and stored in a dictionary. The keys of this dictionary correspond to those of the module table or, for subtables, to those in the already defined list.

```
        data = {
            'column_name': self.config['key']
        }
        content = open(self.source.getTmpPath()).read()
        self.subtable_name_db_value_list = [{'column_name':'entry'}]
        return data
```

When using subtables to store data, the method `fillSubtables()` has to be implemented, executing an SQL command to insert all gathered data into the according subtable.

```
    def fillSubtables(self, parent_id):
        self.subtables['subtable_name'].insert().execute([dict(parent_id=parent_id,
            **row) for row in self.subtable_name_db_value_list])
```

In case subtables were defined in the source code so far, the implementation of the method `getTemplateData()` is necessary in order to make the data accessible from the HTML template.

```
    def getTemplateData(self):
        data = hf.module.ModuleBase.getTemplateData(self)
        details = self.subtables['subtable_name'].select().where(self.subtables['
            subtable_name'].c.parent_id==self.dataset['id']).execute().fetchall()
        data['details'] = map(dict, details)
        return data
```

[20]A free software package for retrieving files using HTTP, HTTPS, and FTP [53].

Using the above sequence of sqlalchemy commands, only relevant data can be selected from the associated subtable. The dictionary derived from this method is accessible from the HTML template.

HTML Template

On the HappyFace web page, the HTML output of all modules is displayed. For the convenient development of the HTML code, HappyFace uses mako templates. This template requires certain elements and provides a number of functionalities, as described in the following.

Initially, the HTML character encoding needs to be specified in the template file. In the world wide web [54], *UTF-8*[21] is the most common encoding. That is why it also used throughout HappyFace.

```
## -*- coding: utf-8 -*-
```

Every HTML template inherits from a base file.

```
<%inherit file="/module_base.html" />
```

The section for the code in the template file is opened as follows:

```
<%def name="content()">
```

In mako templates, embedded Python code can be used. For accessing the information from the module subtables, the key that this information was linked to is used.

```
    % for detail in details:
        <p>${detail['column_name']}</p>
    % endfor
```

In the process of executing the module Python code, all extracted data have been stored in the HappyFace database. The data from the database table corresponding to the module are stored in the variable `module_table`. Lines of embedded Python code are introduced with `%`. In a line that is not embedded Python code, variables are accessed by using the `$` identifier, e. g. `${module.dataset['column_name']}`.

```
    Hello, this is a test ${module.dataset['column_name']}.
```

Finally, the section for the code is closed.

```
</%def>
```

[21] A variable-width encoding for the representation of the Unicode [55] character set [56].

Module Configuration File

To each module belongs a configuration file, which allows for the site-specific adjustment of various parameters. All these parameters for the configuration are given in the module Python source code variable `config_keys`. That is why the generation of a configuration is fairly easy. In this variable, all configuration keys are listed, described, and given a default value. Alongside the HappyFace core, the HappyFace framework provides a set of tools. The `modconfig.py` tool is designed for the automatic generation of module configuration files from the module Python code. The command `python tools.py modconfig MyModule » config/modules-enables/mymodule.cfg` calls the module configuration generation tool and redirects its output to the according module configuration file. Afterwards, the module instance needs to be named properly and a display name for the HappyFace web page should be specified.

Category Configuration File

The HappyFace web page is structured in categories, which are individually configured. For a module to appear on the HappyFace web page, it needs to be assigned to a category. Any category configuration file in the folder `modules/categories-enabled/` serves for this purpose. In this file, the parameter key `modules` needs to be edited and the module instance name added.

Wrap-up

Besides tables that are reserved to the HappyFace core, each module stores its own module table and possibly an arbitrary number of subtables in the HappyFace database. Whenever new modules are added to a HappyFace instance or changes to an existing module were applied, the database schema has to be updated. Therefore, the HappyFace tool `dbupdate` is called. When using the command `yes | python tools.py dbupdate`, all changes to the database schema are confirmed automatically. When all previous steps have been completed, the module is ready for use: `python acquire.py` and `python render.py` can be issued.

4. Web Services

The world wide web was primarily designed for the interaction between humans and applications through hypertext documents via the internet. Three main technologies are the basis of the world wide web: the *Universal Document Identifier (UDI)*, also referred to as URL and *URI*[1], HTML for publishing content, and *HTTP*[2] for the transmission of hypertext documents. For the displaying of hypertext documents, browsers are needed. However, the world wide web as such does not support the interaction of applications with each other.

The interaction of applications through the internet requires web services. These enable applications to expose their services to other applications. The *W3C*[3] defines a WSDL/SOAP-based architecture for web services, as shown in figure 4.1. It involves three parties: a service broker to expose and broker a web service, a service provider, which implements and offers a particular web service, and a service requester, which makes use of the provided web service. The service broker is a registry for web services. Via *UDDI*[4], an XML-based[5] description, web services are registered and located. The

[1] *Uniform Resource Identifier*
[2] *HyperText Transfer Protocol* [2]
[3] *World Wide Web Consortium* [57]
[4] *Universal Description, Discovery and Integration*
[5] *Extensible Markup Language*

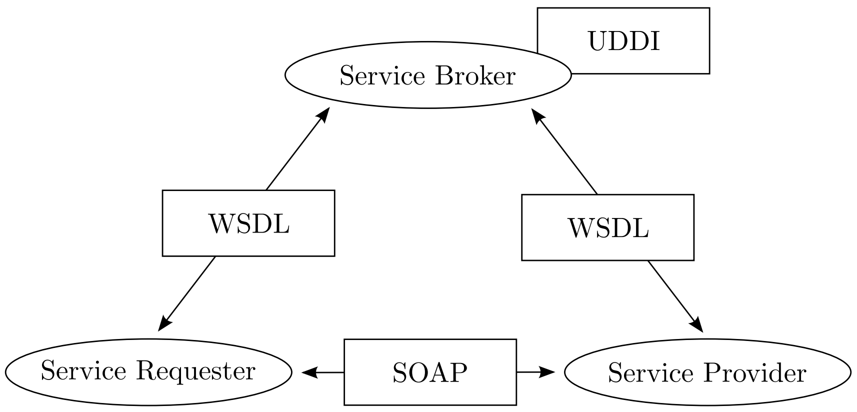


Figure 4.1.: WSDL-based web service architecture as defined by the W3C.

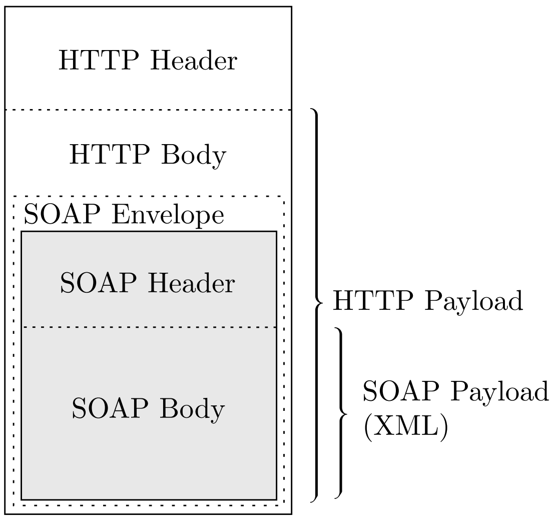


Figure 4.2.: SOAP message as payload of an HTTP transfer.

description of web services are held in the *WSDL*[6] format, which a service provider has offered and a service requester receives. It is also XML-based and specifies a machine-readable description of the location of a web service, the operations it exposes, the data structures involved in communication, and the protocols used for data transmission. For the exchange of information in a structured way, typically *SOAP*[7] is used, which relies on the internet protocol application layer procotol HTTP. Both WSDL and SOAP will be explained in more detail in the following.

Besides WSDL/SOAP-based web services, there are *REST*-based[8] web services. In the architecture of RESTful web services, clients specify requests via URLs. In addition to that, HTTP specifies a set of request methods to manipulate data, e. g. GET, POST, PUT, and DELETE. A server replies to these requests by returning appropriate answers. Important in the concept of REST is the statelessness of communication: The server does not store any state information between two requests, i. e. a request contains all information necessary to service it.

[6] *Web Service Description Language*
[7] *Simple Object Access Protocol*
[8] *Representation State Transfer*

4.1. WSDL/SOAP-based Web Services

4.1.1. Simple Object Access Protocol

SOAP (Simple Object Access Protocol) is a lightweight protocol for the exchange of messages and access to distributed services via a network. Typically, SOAP uses the widely used internet protocol application layer protocol HTTP for the transmission of messages in XML format. A SOAP message consists of a message envelope, the root element of the XML document. This envelope optionally contains a header to describe the processing of the message. The message body is mandatory and contains the actual payload of the SOAP message.

4.1.2. Web Service Description Language

Web services heavily depend on the use of service definitions. Service definitions describe a contract between a service provider and service requesters. WSDL version 1.1 is the most widely used web service description and constitutes a well-defined and recognised standard. In the WCLG community, WSDL web services are widely present, e. g. in dCache and gLite [58].

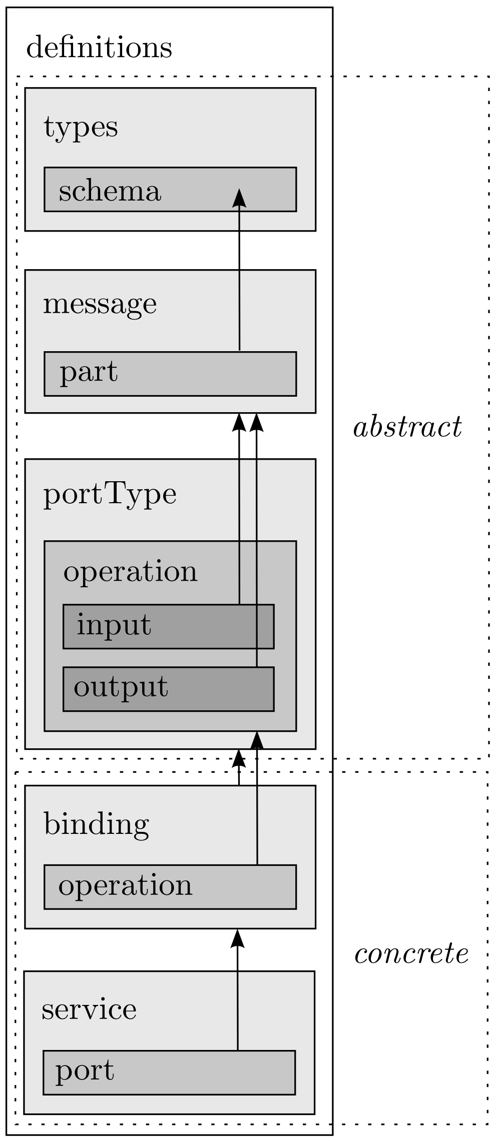


Figure 4.3.: Components and linkages of WSDL 1.1.

WSDL describes the location and syntax of web services using XML. The single XML file that describes the web service is the same for both, the service provider and the service requester. It contains a *definitions* root element, which defines the components *types, message, portType, binding,* and *service.* In addition, a *document* component can be used for documentation purposes. By the use of the *import* component, the web service description can be distributed over multiple documents. While the components *types, message,* and *portType* define the service interface in an abstract way, the components *binding* and *service* describe a concrete implementation. Figure 4.3 shows the WSDL components and their connections.

The WSDL Document Structure

Since WSDL is an XML format, each WSDL document is desired to start with an XML declaration. For interoperability reasons, WSDL documents must use XML version 1.0. Furthermore, the definition of WSDL requires the use of either the UTF-8 or UTF-16 character encoding.

The root element of each WSDL document is the element *definitions*[9]. In the *definitions* element, the name of the WSDL document is set and the XML namespaces[10] used are declared. Furthermore, in the *definitions* element, the elements *types, message, portType, binding,* and *service* are defined. The definition of the elements *import* and *documentation* is optional.

By the *types* element of the WSDL definition, the types section is comprised. In form of a *schema* element, it is a container for user-defined data types that are different from the XML schema built-in types. By the use of built-in types, order indicators, occurrence indicators, and group indicators, arbitrary data structures can be composed. A complex type, which contains an ordered sequence of several built-in data types, is described by the following XML code:

```xml
<element name="monitoring_data">
  <complexType>
    <sequence>
      <element name="grid_site" minOccurs="1" maxOccurs="unbounded" type="xsd:string"/>
      <element name="node_id" type="xsd:string"/>
      <element name="node_status" type="xsd:boolean"/>
      <element name="node_cpu" type="xsd:decimal"/>
      <element name="node_hdd" type="xsd:decimal"/>
    </sequence>
  </complexType>
</element>
```

message elements describe the payload of messages by the use of data elements from the *types* definition. As there can be more than one message, all *message* elements are uniquely named by the use of the *name* attribute. Each message contains one or more *part*

[9]The entire WSDL XML schema is given by [59].
[10]By the use of namespaces, different XML vocabularies can be used and name conflicts are avoided.

elements, which can be compared to the parameters of a function call in a conventional programming language.

```
<message name="RequestMonitoringData">
  <part name="grid_site" type="xsd:string"/>
  <part name="node_id" type="xsd:string"/>
</message>
<message name="ReturnMonitoringData">
  <part name="data" type="tns:monitoring_data"/>
</message>
```

A *portType* defines an abstract interface for a web service and relates the *input* and *output* of *operation* elements to *message* elements. There can be more than one *portType* element. The use of multiple *portType* elements allows for a grouping of related operations. Each *portType* can contain an arbitrary number of *operation* elements.

```
<portType name="MonitoringPort">
  <operation name="GetMonitoringData">
    <input name="RequestMonitoringData" message="RequestMonitoringData"/>
    <output name="ReturnMonitoringData" message="ReturnMonitoringData"/>
  </operation>
</portType>
```

No concrete transmission protocol and encoding style has been defined in the elements *types, message,* and *portType.* The *binding* element specifies the details of data transmission by substantiating the abstract definitions. Each (named) *binding* elements binds *portType* and *operation* elements to a specific protocol, e. g. SOAP. For a SOAP binding, the elements *soap:binding, soap:operation*, and *soap:body* are used. The *soap:binding* element defines a service as a SOAP service, the element *soap:body* indicates that the message data are sent in the SOAP message body. The element *soap:operation* is optional and used to identify SOAP requests that invoke the specified operation via the attribute *soapAction.*

```
<binding name="MonitoringPortBinding" type="MonitoringPort">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetMonitoringData">
    <soap:operation soapAction="urn:monitoring.wsdl#GetMonitoringData"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

Each *service* element is a group of *port* elements and associates a *binding* element with an access address. Thus, each *service* element constitutes a web service. Different ports can be used to access different logically grouped parts of a web service. In the *port* element,

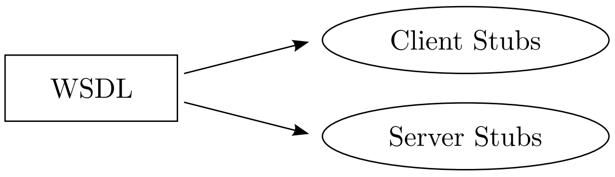


Figure 4.4.: Stub generation from WSDL.

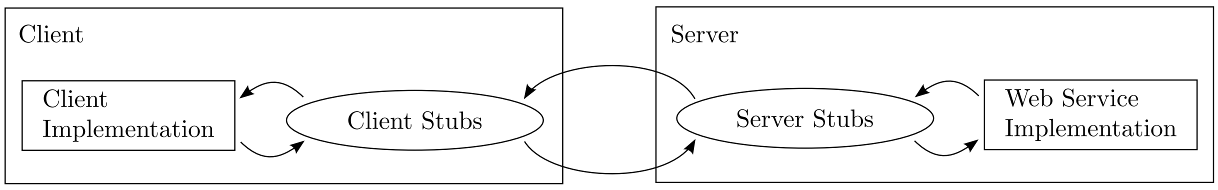


Figure 4.5.: Client and server using the stubs generated from WSDL.

the access address is provided by the *soap:address* extension element when the protocol used is SOAP.

```
<service name="MonitoringService">
  <port name="MonitoringServerPort" binding="MonitoringPortBinding">
    <soap:address location="http://example.com/monitoringdata"/>
  </port>
</service>
```

4.1.3. Stub Generation

A WSDL document specifies the public interface to a web service. Nevertheless, it does not define what the service requester and service provider actually do. Service requesters and providers are free in their implementation, as long as their messages conform to the service definition. There are libraries for all common programming languages, which allow for the generation of service requester (client) and service provider (server) *stubs* from WSDL documents, as shown in figure 4.4. Stubs are abstract adapters for data types, methods and interfaces to the web service. These automatically created documents are then used both by the client and the server implementation, see figure 4.5.

4.1.4. WSDL/SOAP-based Web Services in Python

The Python *ZSI*[11] web services toolkit is designed for the implementation of web services in Python and supports the WSDL/SOAP-based web service architecture. By the use of the tool `wsdl2py`, client and server stubs are generated from a WSDL document. It creates Python bindings for the services and data structures and provides a server skeleton for

[11] *Zolera SOAP Infrastructure* [60]

HTTP Method	Description
GET	Resource retrieval via the specification of a URI. In the URI, arguments can be passed to the server.
POST	Resource creation via the specification of a URI. Transmission of data either as part of the URI or in the body of the HTTP message.
PUT	Upload of a resource by specification of a target URI.
DELETE	Deletion of a specified resource.

Table 4.1.: Most notable HTTP request methods [2].

the service dispatch. In order to create a running web service server from this, an HTTP server and a request handler need to be implemented. On client side, the appropriate web service needs to be located, the services bound, and the request data structures need to be populated. After an operation was called, the returned data structures are read.

4.2. RESTful Web Services

A simple alternative to WSDL/SOAP-based web services are RESTful web services. Since these web services are easy-to-consume and need less overhead, they are widely used. RESTful web services are non-standardised, do not provide a description of their functionalities, and there is no registry for the advertisement of their existence. There are several main principles that RESTful web services are required to abide to. Most importantly, RESTful web services are stateless. This simplifies the server-side implementation and leaves the client implementation the responsibility for the maintenance of state information. Furthermore, RESTful web services make use of HTTP methods. Table 4.1 contains a description of the most notable HTTP request methods in terms of web services. Another principle is the use of directory structure-like URIs. These URIs are hierarchical, splitting up in sub-branches from a root branch. This ensures the intuitive usability of the web service, especially for the use of exposed resources. Finally, RESTful web services should be able to transfer resources in XML or JSON[12] format. By the use of MIME-types[13] and the HTTP accept message header, a mechanism called *content negotiation* [61] can be used in order to determine the desired data format.

[12] *JavaScript Object Notation*
[13] *Multipurpose Internet Mail Extensions*

4.3. Web Services for Access to the HappyFace Database

All data present on the HappyFace web page are stored in the HappyFace database. By using the history functionality of HappyFace, data for previous points in time can be displayed on the HappyFace web page. However, the HappyFace data are not designed for access by applications over a network. For HappyFace, data retrieval is only possible via direct database access on the machine where the HappyFace framework is located.

4.3.1. Database Server vs. Web Service

In principle, the HappyFace database could be made available via a database server. A database server provides local databases to remote applications via a computer network. Most database servers also offer database management functionalities and have a high functional range. Typically, data base servers also handle authentication, caching, and tuning mechanisms for databases. There is a large number of commercial and non-commercial database server systems, each using a different interface for data queries and data retrieval. Even though some of these interfaces are accessible by applications, there is no established standard for the connection to a database, the submission of queries, and the retrieval of data. Furthermore, the retrieved data structures are not well-defined and access is not restricted to specific operations only, since database servers are designed for general access to databases. By the use of a W3C-compliant web service, the connection to the provided interface, the data structures, and the permitted operations are well-defined and allow for a much more precise and focused data access. As defined in the WSDL document, which fully describes the web service, only relevant data retrieval operations are permitted and prevent high load on this critical part of the HappyFace framework. Most important, the use of a web service for database access does not require the full understanding of the underlying data(base) schema. Regardless of HappyFace-specific structures, data can be obtained in a machine-readable format for further processing.

4.3.2. A WSDL/SOAP-based Web Service for Access to the HappyFace Database

The W3C standard for WSDL/SOAP-based web services allows for the access to the HappyFace data regardless of HappyFace-specific data structures. Tools for the automatic stub generation facilitate the easy implementation of client and server applications, fully independent of the platform that client and server are run on.

Python WSDL Generator

A WSDL file is a static document. However, the manual adaptation of the WSDL document to changes in the HappyFace database, e. g. new tables, new table columns, or table alterations, is not desirable. Therefore, a WSDL generator was implemented. It reads the HappyFace database structures and generates a WSDL file, not requiring human intervention. The execution of the WSDL generator requires access to the WSDL database. Via sqlalchemy, an object-relational mapping is performed. Thereby, all database tables and their columns are mapped to Python objects, which provide member functions for data retrieval, manipulation, etc. By the use of these Python objects, the WSDL file is generated.

WSDL Document Layout

The WSDL document contains the required root element *definitions* and its sub-elements *types, message, portType, binding,* and *service.* For the specification of the time interval the data are requested from, a data type is assembled from the *xsd:datetime* type. Each table in the database requires two data types in the WSDL *types* definition. A data type for the request of information from a specific table is a sequence of *xsd:boolean* variables. There is one boolean variable for each table column, which specifies whether a column is requested or not. As the occurrence of these boolean variables is optional, a client request only needs to specify the columns it wants to retrieve. The data type for the response is a sequence of variables, each of which is a list. It is supposed to contain all table entries from the specified time range for a specific table column.

The WSDL *message* element can occur multiple times in a WSDL document. For each table from the HappyFace database, there are two messages. The first message contains the request data structure and consists of three parts: a data structure for the module instance name, a time interval data structure, and the list of boolean variables defined for the request. The second method transmits the response data structure and uses the according data type defined beforehand for the response.

There is one *portType* element for each database table. It defines the *operation* for the request and retrieval of data. Its input is the request *message* element, its output the response *message* element. By defining one *portType* element per table, the data are well-separated in the WSDL definition.

When a *portType* element was defined, it needs to be bound to a transmission protocol. For all *portType* elements, the SOAP transmission method is chosen, as provided by the W3C standard definition. There is one WSDL *binding* element for each database table.

The *service* element finally defines a web service. It assembles *binding* elements and defines concrete access ports for the different *binding* elements. For access to the web service via SOAP, *soap:address* elements are defined.

In the appendix A.4 of this document, a simplified excerpt from the WSDL document described is given.

Client and Server Implementation

Using the tool *wsdl2py*, which comes with the Python ZSI toolkit, the WSDL document is used to generate the stubs for server and client. A Python client and server implementation use the stubs and grant access to all HappyFace data.

A WSDL documents does not define the concrete implementation of client and server. First of all, the server assesses the received request. If not all mandatory variables are defined, it returns an error code and terminates the connection to the querying client. If the query matches the WSDL definition, the server implementation retrieves requested data by querying the HappyFace database. In that respect, an object-relational mapping is performed. The data structures defined in the WSDL file are not a one-to-one representation of the data structures in the HappyFace database. Each table in the database is mapped to a response data structure, which also contains information to uniquely identify a dataset. This information is only available in the general table *hf_runs* and requires the following SQL query structure:

```sql
SELECT hf_runs.time, hf_runs.id, table1.col1 FROM hf_runs INNER JOIN table1 on hf_runs.
    id = table1.id WHERE [timestamp_from] < hf_runs.time AND [timestamp_to] > hf_runs.
    time;
```

Please refer to chapter 3.3.1 for details on the HappyFace database.

For the use of a ZSI web server, the server implementation needs to provide a Python class for each service port, which inherits from a base class defined in the stub files. Each service port class defines a method for the chosen SOAP binding and operation. This method returns, amongst others, a response object that contains member variables for simple data types and Python class instances for all complex data types defined in the WSDL file.

The implementation of a Python client was performed for testing purposes and to provide a working code to interested parties. A client request is required to match the WSDL definition. In the Python ZSI framework, a request object needs to contain class instances of all complex element classes. There is a class for the specification of the time interval and a class for the specification of the requested database table columns. The returned response data structure contains a list for each column queried. These iterable objects allow for easy further processing of the data.

4.3.3. A RESTful Web Service for Access to the HappyFace Database

A RESTful web service for access to the HappyFace database requires the knowledge of the available database table columns. Via the call of a composed URL, arbitrary module tables and subtables can be queried. The result of any query is returned in JSON format. This web service is also platform-independent.

Python CGI Script

The Python script implementing the RESTful web service is a CGI[14] script and expects the specification of its input parameters via a composed URL as follows:

```
http://path/to/script.py?table_name=table1&instance=instance_name&columns=col1,col2&
    table_name=table2&columns=col3,col4&timestamp_from=YYYY-MM-DD HH:mm&timestamp_to=YYYY
    -MM-DD HH:mm
```

Missing or faulty parameters are detected automatically and the user is notified. In case of a correctly composed request, the HappyFace database is queried. The result of this query is translated into JSON format and returned to the service requester.

4.3.4. Response Time Comparison

A key benchmark for web services is their total response time. For the comparison of the average response time of the WSDL/SOAP-based and the RESTful web service for access to the HappyFace database, queries of different numbers of return values are sent

[14] *Common Gateway Interface*

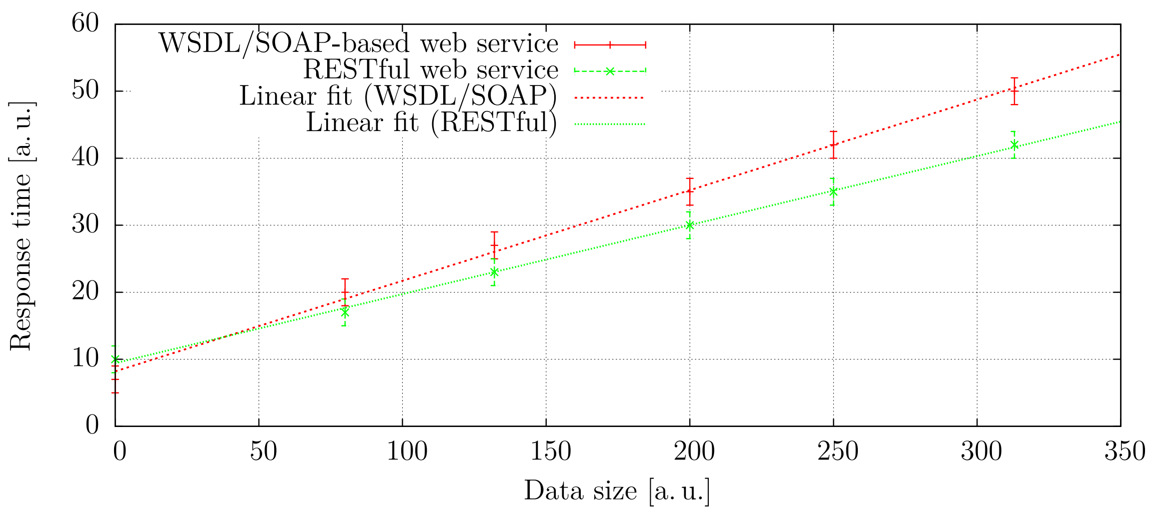


Figure 4.6.: Response time comparison of WSDL/SOAP-based and RESTful web service for access to the HappyFace database. Both web services exhibit a linear behaviour.

to both web services. In order to guarantee comparability, both client and server are run on the same computer for both web services. Finally, an averaging over multiple, identical requests provides higher statistical certainty. In figure 4.6, the average response time is plotted versus the number of database entries returned.

In the region considered, both web services exhibit linear behaviour. This satisfies the expectation, as all main steps – the extraction of data from the database, the encapsulation of data in packages, and the transfer of data – are operations, whose complexity is approximately linear. In both cases, the ordinate intercept is negligible for high numbers of return values. The difference in ordinate intercepts probably results from the different web servers used for providing the web service.

Both, the WSDL/SOAP-based and the RESTful web service, perform well for their expected use cases. While the WSDL web service provides a fully standardised access to the HappyFace database but is slower in the provisioning of responses, the RESTful web service shows fast responses but its usage requires a better knowledge of the queried data. Both web services extend the range HappyFace functionalities and make HappyFace monitoring data easily available.

5. HappyFace Module Development for GoeGrid

The HappyFace framework strictly separates its core part from the individual modules. Each module is designed to retrieve data from one source or a set of associated resources, analyse, assess, and present this information. In the course of decommissioning HappyFace version 2 and putting HappyFace version 3 into production at the WLCG ATLAS Tier-2 centre GoeGrid and other sites using HappyFace, all relevant modules required redevelopment. Due to the fact that some of the old modules have not proven useful, these modules were decommissioned completely. All modules that have been in active use were redeveloped and experienced improvements in terms of flexibility and configurability. In the instance of HappyFace 2, several pieces of critical information were not assessed. This required the development of completely new modules, which cover these monitoring sources and allow for an easier prevention and better understanding of GoeGrid failures. All modules developed are designed to be generic and are adaptable to any ATLAS grid site in the WLCG just by the adaptation of the according module configuration file.

Besides the HappyFace modules described in this chapter, several dCache modules were adapted to the dCache version run in GoeGrid. For these modules, please refer to appendix A.5.

5.1. HammerCloud Functional Tests

The *HammerCloud* tests are a distributed analysis testing service. Their goal is to measure site performance using real jobs. In the first instance, HammerCloud tests were designed as stress tests, heavily involved in the commissioning of new sites. HammerCloud were also used for site infrastructure and software changes evaluation and the comparison of site performances. Since the WLCG has reached a steady state of operation, the main focus of the HammerCloud tests has shifted towards continuous validation testing. Functional tests perform site validation with continuous streams of real jobs. Three *Virtual Organi-*

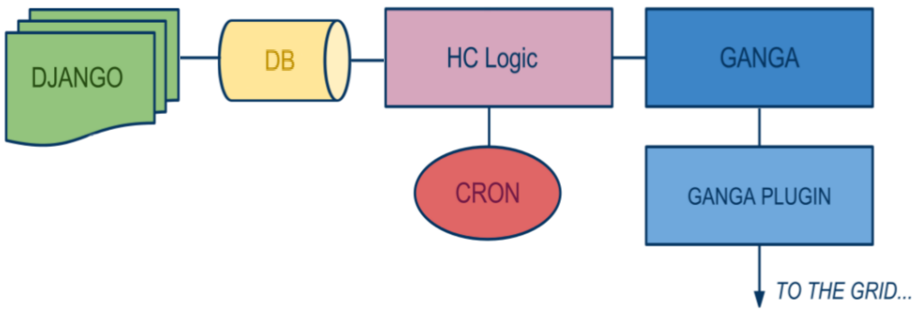


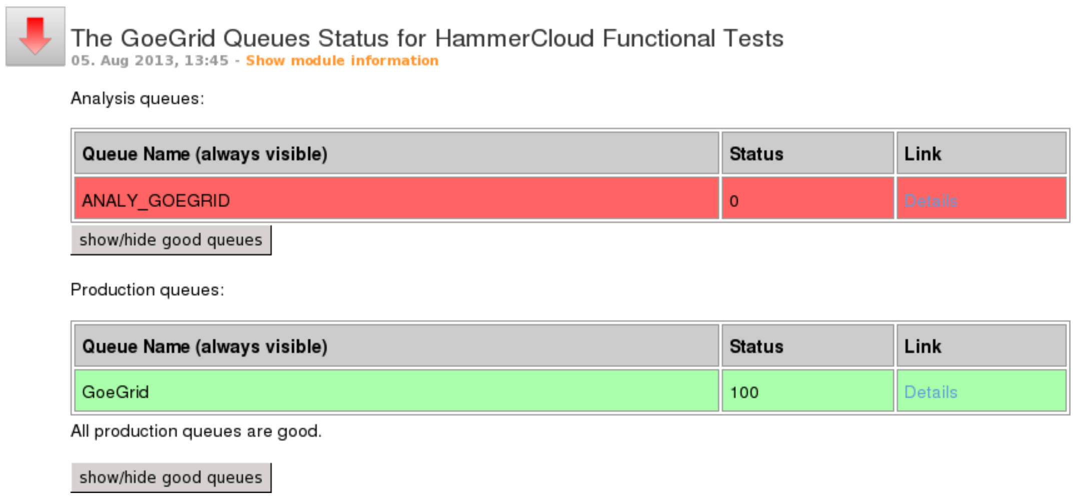
Figure 5.1.: Architecture of the HammerCloud tests service [1]. The *Django*-based front-end publishes test results in human- and machine-readable format.

sations (VOs) use the HammerCloud functional tests: ATLAS, CMS, and LHCb. For the VO ATLAS, there are functional tests for user analysis and Monte Carlo simulation. In case a certain number of these tests fails, a PanDA queue is automatically excluded for further job submission. Nevertheless, HammerCloud tests are launched and allow the site to be available for submission when the test performance has improved again. In figure 5.1, the HammerCloud architecture is shown. Via a web front-end, the test results can be accessed and obtained in JSON format for further processing.

For each queue specified in the module configuration file, a status value is obtained from the HammerCloud web front-end. The status, indicating the success rate of the HammerCloud tests, is only rated *ok* when all test jobs were successful. By default, all monitored queues that were rated *ok*, are hidden and can be expanded. When specified in the module configuration file, particular queues can be set visible permanently, regardless of their status. In figure 5.2, the module output is displayed.

5.2. Compute Node Information

Each involved grid site provides computing resources to the WLCG infrastructure. Besides storage resources, a grid site also contributes to WLCG operations by the allocation of compute nodes. On these Linux machines, user analysis and Monte Carlo simulation jobs are executed. During the process of execution, a job is assigned several states by PanDA. The regular sequence of states is the following [62]: *pending* → *defined* → *waiting* → *assigned* → *activated* → *sent* → *starting* → *running* → *holding* → *transferring* → *finished/failed/cancelled*. On the PanDA monitoring web page, this information is published. For each compute node, the number of jobs in a specific state is given.


The GoeGrid Queues Status for HammerCloud Functional Tests
05. Aug 2013, 13:45 - Show module information

Analysis queues:

Queue Name (always visible)	Status	Link
ANALY_GOEGRID	0	Details

show/hide good queues

Production queues:

Queue Name (always visible)	Status	Link
GoeGrid	100	Details

All production queues are good.

show/hide good queues

Figure 5.2.: HTML output of the HammerCloud functional tests module. At GoeGrid, the analysis queue *ANALY_GOEGRID* and the production queue *GoeGrid* are monitored. The status of the analysis queue is rated *critical* due to the insufficient success rate of the HammerCloud tests.

In order to evaluate the functionality of a compute node, especially information about the share of failed jobs is important. A high failure rate may indicate a problem and require the exclusion or restart of a worker node. The compute node information module identifies the compute nodes with the highest failure rate and displays them on first sight. A list containing all compute nodes can be expanded. For fast visual identification of such compute nodes that exhibit an abnormally high failure rate, all compute nodes that show job failures are presented in a bar graph. A mouse-over text serves for the identification of the compute node name and the exact number of failures. When the total number of transferring jobs is large, this may induce a high load on the storage management and can be the cause of failures. By the identification of worker nodes that run a high number of transferring jobs, problems can be foreseen. In an HTML table, this module shows the compute nodes with the largest number of transferring jobs. A part of the HTML output of this module is shown in figure 5.3.

This module also serves the purpose of identifying *black hole* worker nodes. Black holes are empirically defined as worker nodes that are able to accept jobs but cause the immediate failure of the accepted jobs. This may be due to the failure of a critical service or operating system component while the PBS client is still functional and accepts jobs. For the PBS server, this worker node looks perfectly functional and is even a preferred node for the processing of new jobs due to its low load. As a result of this, a high number of jobs fails. Therefore, black hole worker nodes need to be identified immediately.

Typically, one site provides more than one PanDA queue. In the configuration of this module, multiple queue names can be specified. Furthermore, thresholds for a *warning* and *critical* status can be set with regard to the job failure and transferring rate.

5.3. Analysis GANGA Jobs

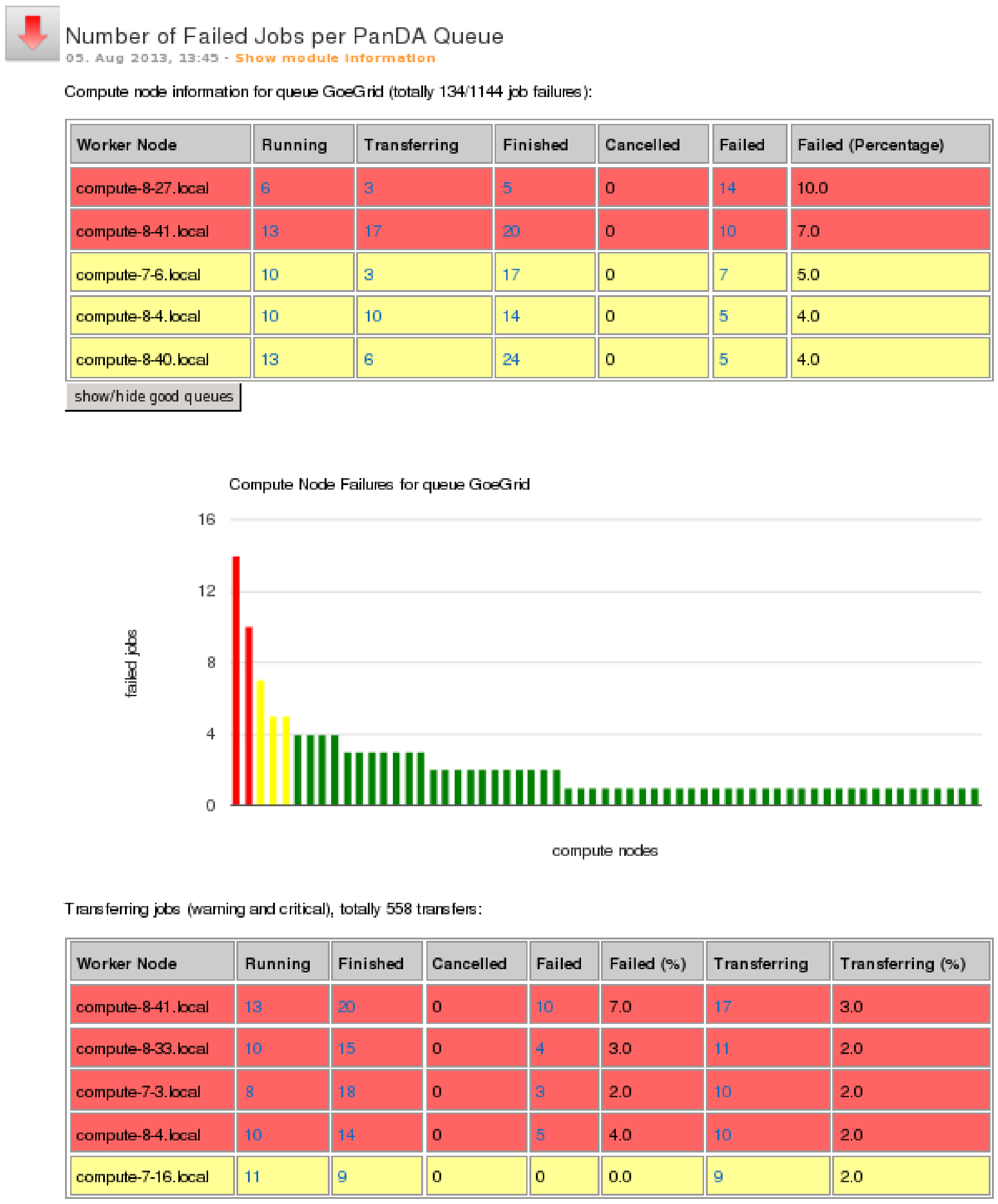
GANGA[1] is a user front-end for job submission to the WLCG. It is a collaborative development of ATLAS and LHCb and covers all phases of a job: creation, configuration, splitting, reassembly, script generation, file transfer, submission, run-time setups, monitoring, and reporting. Via GANGA, HammerCloud test jobs are sent using a special certificate. The PanDA monitoring web page provides details on these jobs. For an ATLAS WLCG site, the sequential failure of three out of four tests jobs leads to the exclusion of a queue and must therefore be monitored.

In the Python source code of this module, the Panda HTML web page is parsed and information on GANGA robot jobs is extracted. By the relative number of failed jobs, the gathered information is assessed. According to predefined thresholds, a certain share of failed GANGA robot jobs leads to a warning or a critical status of this module, indicated by a colouring schema. In the configuration of this module, queues can be specified that are always visible in the HTML result table. Otherwise, all queues not rated as warning or critical are hidden and can be expanded. In figure 5.4, the module web output is displayed.

5.4. Nagios Monitoring

Nagios is a generic, open-source monitoring system, supporting system monitoring, network monitoring, and infrastructure monitoring. Designed for generic monitoring, Nagios is used to monitor servers on which WLCG services are run. Of specific interest for each service are certain monitoring parameters that are captured by Nagios. All these parameters are collocated on a status web page. In most setups of Nagios, authentication is required in order to access this summary web page. For the sake of convenience and simplicity, Nagios applies a configurable rating to each parameter. In terms of real-time monitoring, it can send alerts in case critical infrastructure fails.

[1] *Gaudi/Athena and Grid Alliance* [63]



Number of Failed Jobs per PanDA Queue

05. Aug 2013, 13:45 - Show module information

Compute node information for queue GoeGrid (totally 134/1144 job failures):

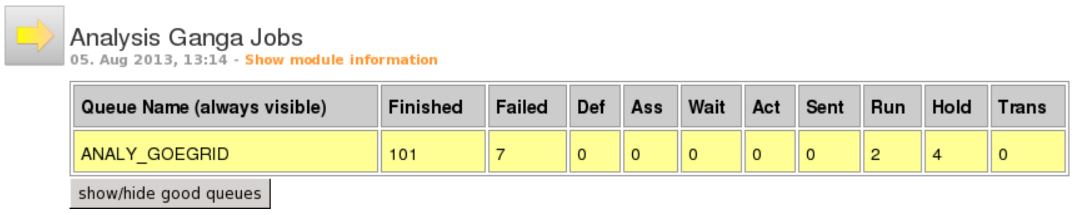
Worker Node	Running	Transferring	Finished	Cancelled	Failed	Failed (Percentage)
compute-8-27.local	6	3	5	0	14	10.0
compute-8-41.local	13	17	20	0	10	7.0
compute-7-6.local	10	3	17	0	7	5.0
compute-8-4.local	10	10	14	0	5	4.0
compute-8-40.local	13	6	24	0	5	4.0

show/hide good queues

Transferring jobs (warning and critical), totally 558 transfers:

Worker Node	Running	Finished	Cancelled	Failed	Failed (%)	Transferring	Transferring (%)
compute-8-41.local	13	20	0	10	7.0	17	3.0
compute-8-33.local	10	15	0	4	3.0	11	2.0
compute-7-3.local	8	18	0	3	2.0	10	2.0
compute-8-4.local	10	14	0	5	4.0	10	2.0
compute-7-16.local	11	9	0	0	0.0	9	2.0

Figure 5.3.: View of the compute node information module in production for GoeGrid. The module contains two tables and one bar graph for each PanDA queue specified in the module configuration file.


Analysis Ganga Jobs
05. Aug 2013, 13:14 - Show module information

Queue Name (always visible)	Finished	Failed	Def	Ass	Wait	Act	Sent	Run	Hold	Trans
ANALY_GOEGRID	101	7	0	0	0	0	0	2	4	0

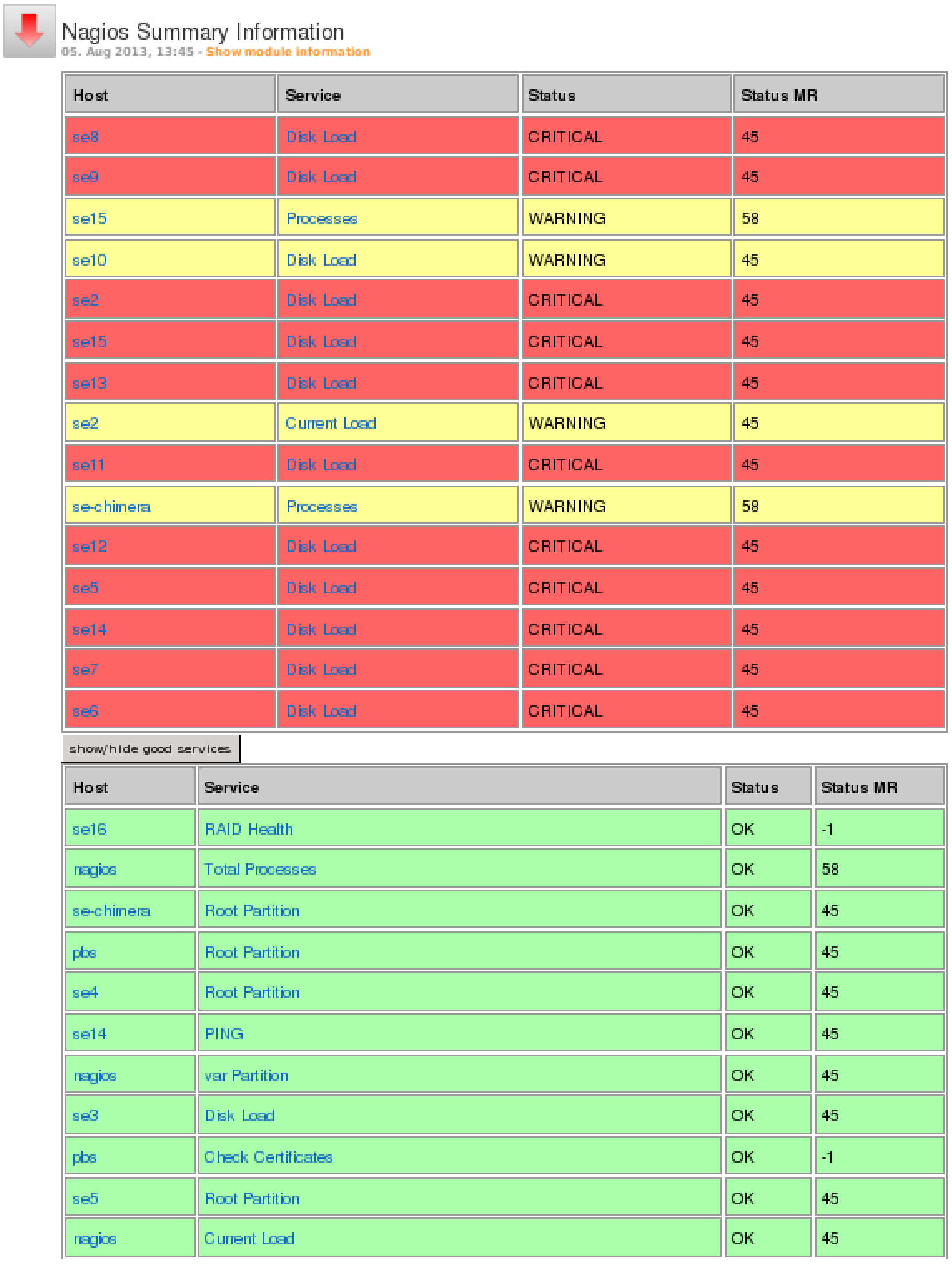
Figure 5.4.: View of the analysis GANGA jobs monitoring module HTML output. Yellow table rows indicates a *warning* status, a *critical* status is indicated by a red colouring. For monitoring at GoeGrid, the queue *ANALY_GOEGRID* is always visible.

As the location of Nagios is most likely different for each grid site, it needs to be configurable in the module configuration file. Besides that, it is proposed to create an additional Nagios user. For obvious security reasons, this user should be equipped with no more than read rights in the Nagios system. This is due to the fact that both user name and password have to be specified in the unencrypted HappyFace module configuration file. The Nagios status web page displays the information in an HTML table, which is downloaded and parsed by the HappyFace module. In a next step, relevant data are extracted and stored in the HappyFace database. From this database, the content is read and displayed in a table format. HappyFace is designed to detect problems on first sight. Due to that, only such services and parameters are displayed, which require further investigation, such as warnings and failures. By clicking one button, this view can be extended to show all other parameters, too. Being generically designed and easily configurable, this module might be the subject of interest for any WLCG site using Nagios and deploying or planning to deploy HappyFace.

5.5. Ganglia Monitoring

Ganglia monitors cluster hardware and software and is in wide use in various high-performance computing centres. The scalable and distributed system provides real-time monitoring, e. g. of CPU loads, memory usage, and network traffic. On each monitored node, a Ganglia daemon needs to be installed. A web front-end compiles the acquired information. Besides this human-readable interface, Ganglia provides all acquired information in an XML format for further automated processing.

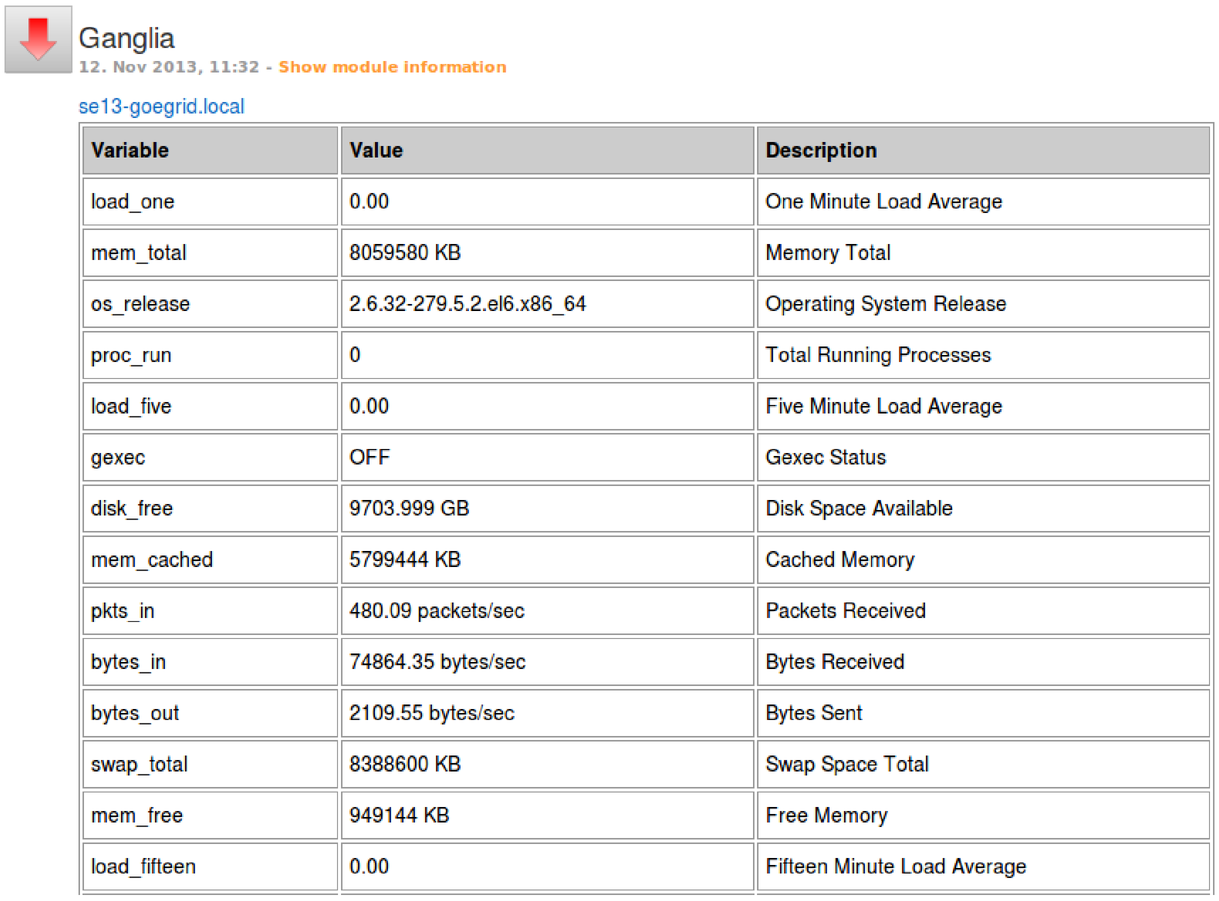
All GoeGrid servers, such as storage servers, the rocks server, the Nagios server, the CREAMCE server, the BDII server, and the servers hosting virtual machines, are monitored by Ganglia. It records parameters such as the load of these servers, the memory


Nagios Summary Information
05. Aug 2013, 13:45 - Show module information
show/hide good services

Host	Service	Status	Status MR
se8	Disk Load	CRITICAL	45
se9	Disk Load	CRITICAL	45
se15	Processes	WARNING	58
se10	Disk Load	WARNING	45
se2	Disk Load	CRITICAL	45
se15	Disk Load	CRITICAL	45
se13	Disk Load	CRITICAL	45
se2	Current Load	WARNING	45
se11	Disk Load	CRITICAL	45
se-chimera	Processes	WARNING	58
se12	Disk Load	CRITICAL	45
se5	Disk Load	CRITICAL	45
se14	Disk Load	CRITICAL	45
se7	Disk Load	CRITICAL	45
se6	Disk Load	CRITICAL	45

Host	Service	Status	Status MR
se16	RAID Health	OK	-1
nagios	Total Processes	OK	58
se-chimera	Root Partition	OK	45
pbs	Root Partition	OK	45
se4	Root Partition	OK	45
se14	PING	OK	45
nagios	var Partition	OK	45
se3	Disk Load	OK	45
pbs	Check Certificates	OK	-1
se5	Root Partition	OK	45
nagios	Current Load	OK	45

Figure 5.5.: View of the Nagios monitoring module HTML output. Yellow table rows indicates a *warning* status, a *critical* status is indicated by a red colouring. For testing purposes, the machine-readable table column is displayed. At the time of this screenshot, the by default hidden table showing all good services is expanded.


Ganglia
12. Nov 2013, 11:32 - Show module information

se13-goegrid.local

Variable	Value	Description
load_one	0.00	One Minute Load Average
mem_total	8059580 KB	Memory Total
os_release	2.6.32-279.5.2.el6.x86_64	Operating System Release
proc_run	0	Total Running Processes
load_five	0.00	Five Minute Load Average
gexec	OFF	Gexec Status
disk_free	9703.999 GB	Disk Space Available
mem_cached	5799444 KB	Cached Memory
pkts_in	480.09 packets/sec	Packets Received
bytes_in	74864.35 bytes/sec	Bytes Received
bytes_out	2109.55 bytes/sec	Bytes Sent
swap_total	8388600 KB	Swap Space Total
mem_free	949144 KB	Free Memory
load_fifteen	0.00	Fifteen Minute Load Average

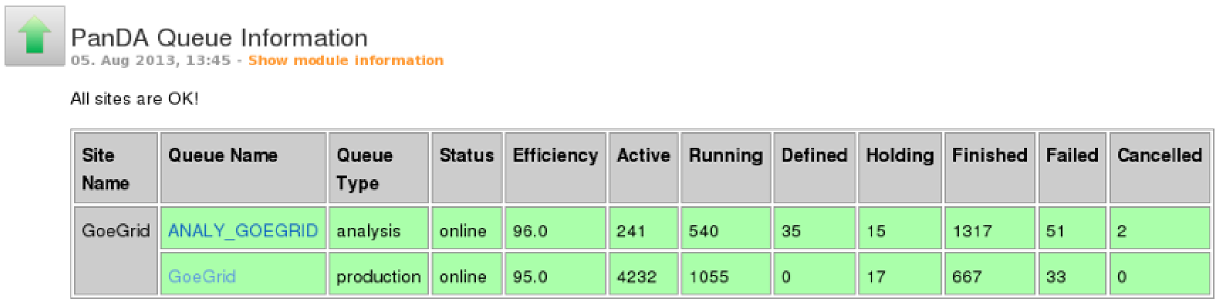
Figure 5.6.: Web output of the Ganglia monitoring module. All monitored servers specified in the module configuration file are listed separately, all presented data are expanded on-click.

and swap usage, the available disk space, and its network activities. Via its XML output, HappyFace acquires this information and stores it to the HappyFace database. When displaying the information, grouped by host and presented in expandable tables, no rating takes place by default. This is due to the fact that the criticality of parameters does not only depend on its present value but on its development and all other parameters as well.

The module is designed for generic use and can be adapted by other sites by the change of the module configuration file only. In figure 5.6, its HTML output is shown.

5.6. PanDA Monitoring

The most important component of the ADC infrastructure is the PanDA production system. It constitutes the management role for the execution of all ATLAS production and


PanDA Queue Information

05. Aug 2013, 13:45 - Show module information

All sites are OK!

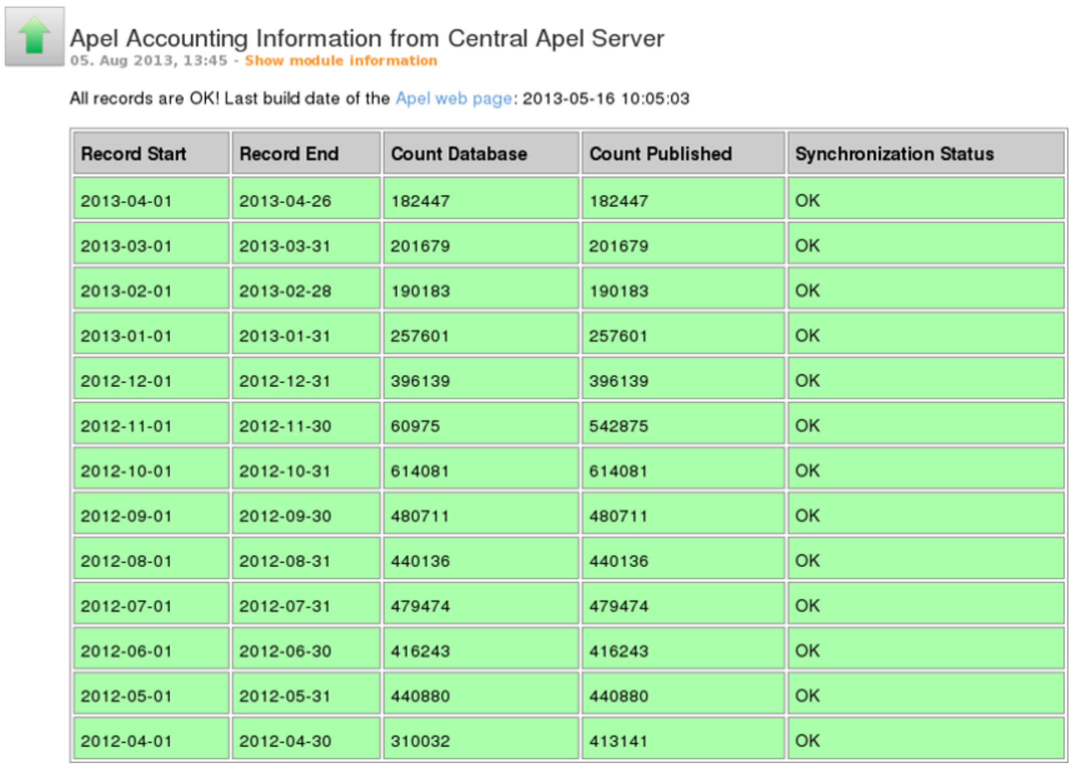
Site Name	Queue Name	Queue Type	Status	Efficiency	Active	Running	Defined	Holding	Finished	Failed	Cancelled
GoeGrid	ANALY_GOEGRID	analysis	online	96.0	241	540	35	15	1317	51	2
	GoeGrid	production	online	95.0	4232	1055	0	17	667	33	0

Figure 5.7.: HTML output of the PanDA monitoring module. As this module only relies on external monitoring sources, it is fully independent of local resources.

a significant share of the user and group analysis jobs. PanDA is also a core component of the dataset replication system and plays a key role in the definition of production tasks as well. Nowadays, PanDA is a unified production system for ATLAS across the *Open Science Grid (OSG)* [64], EGI, and *Nordic DataGrid Facility (NDGF)* [65]. In the PanDA system, two types of virtual queues exist. They are defined according to the types of jobs running on a site. If an ADC site is able to accept and process user and group analysis, as well as ATLAS Monte-Carlo simulation jobs, it has both an *analysis* and a *production* queue, managed automatically. Depending on the efficiency of the submitted jobs and whether the site is in a scheduled or unscheduled downtime, an *online*, *offline*, or *test mode* state is assigned to the queues. If no downtime is declared and the queue efficiency drops below a certain level, a queue is switched to test mode. From that moment on, only test jobs are submitted to the queue, and the reliability and availability of this queue are negatively affected. As soon as the efficiency reaches a certain level again, the site is switched online. Only during downtimes, queues are in offline mode. Taking these facts into account, it is extremely important to keep an eye on the queue efficiency and the PanDA queue status of a site.

Except for the data processing at the CERN Tier-0 computing centre, the PanDA production system is used ATLAS-wide for simulation, reprocessing, and production of physics data. In order to determine whether a site has available resources, *pilot jobs* are sent to all grid sites. Furthermore, they guarantee the availability of resources for real user jobs. In order to determine a site's availability and reliability, *test jobs* are sent. These tests also contribute to the determination of the overall queue status.

In HappyFace version 2, two modules were developed to show all relevant information from the PanDA system. One of them displays all activated, assigned, running, transferring, holding, finished, and failed jobs. A second module shows the overall status of


Apel Accounting Information from Central Apel Server
05. Aug 2013, 13:45 - Show module information

All records are OK! Last build date of the Apel web page: 2013-05-16 10:05:03

Record Start	Record End	Count Database	Count Published	Synchronization Status
2013-04-01	2013-04-26	182447	182447	OK
2013-03-01	2013-03-31	201679	201679	OK
2013-02-01	2013-02-28	190183	190183	OK
2013-01-01	2013-01-31	257601	257601	OK
2012-12-01	2012-12-31	396139	396139	OK
2012-11-01	2012-11-30	60975	542875	OK
2012-10-01	2012-10-31	614081	614081	OK
2012-09-01	2012-09-30	480711	480711	OK
2012-08-01	2012-08-31	440136	440136	OK
2012-07-01	2012-07-31	479474	479474	OK
2012-06-01	2012-06-30	416243	416243	OK
2012-05-01	2012-05-31	440880	440880	OK
2012-04-01	2012-04-30	310032	413141	OK

Figure 5.8.: Web output of the APEL accounting module. This module only relies on external monitoring sources and therefore is fully functional independent of the system it is deployed on, as long as it is connected to the world wide web.

all GoeGrid queues. For this new module, the monitoring information is retrieved from two types of sources. In order to obtain the number of jobs running on GoeGrid and their status, an HTML table is analysed and relevant data are extracted using *regular expressions*. The information on the queue status is available in JSON format. In the module configuration file, user defined-thresholds are set, according to which the queue status is rated and a colouring schema is applied.

The newly developed module merges these information into one table and allows for a simplified check of the PanDA queues. Furthermore, it combines and clearly arranges queues that belong to the same grid site.

5.7. APEL Accounting

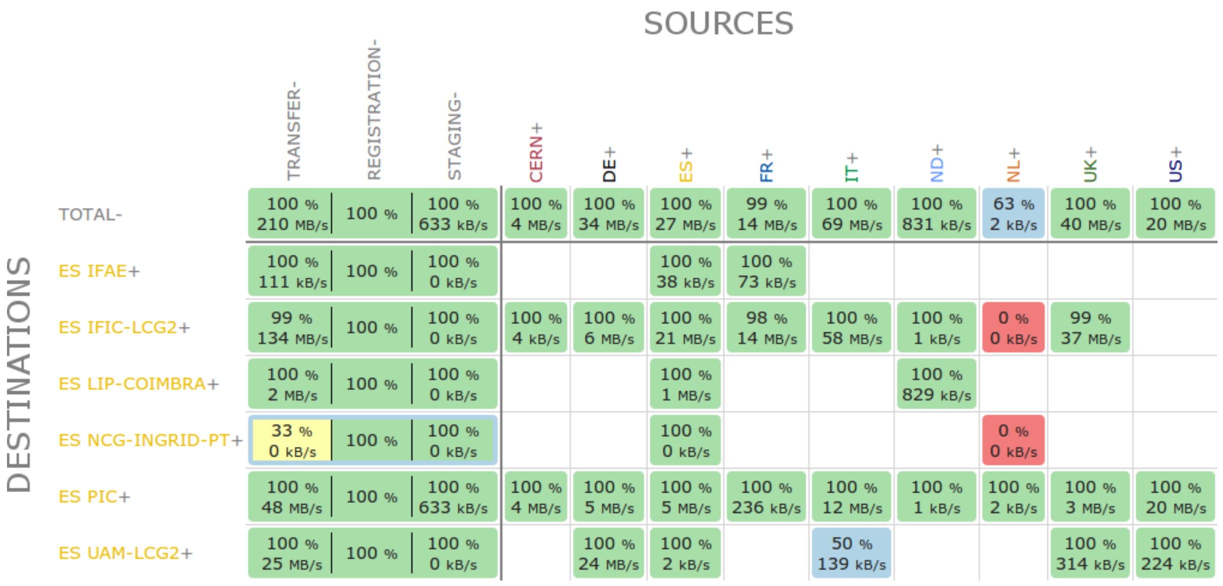
APEL is responsible for publishing the accounting data of a grid site, registering the amount of resources spent on grid jobs to the central accounting server. For some period of time, APEL accounting information happened not to be published properly, due to misconfiguration in GoeGrid and problems of the central APEL accounting server. This lead to wrong numbers in the WLCG accounting reports. Therefore, the periodic publishing of accounting information needs to be monitored, in order to avoid failures in up-to-date accounting information publishing.

The results of the APEL synchronisation tests are available via a web interface, formatted as an HTML table. This table is read in and the latest accounting information entries are extracted. The most recent entries are displayed on the HappyFace web page in an HTML table. During the processing of information, the age of these entries is compared to thresholds defined in the module configuration file. Based on this, the module status is determined and a colouring schema is applied to the rows of the generated table. The web output of this module is shown in figure 5.8.

5.8. DDM Dashboard Monitoring

The ATLAS DDM, alongside the PanDA system, plays a key role in the ATLAS computing model. It organises more than 90 PBytes of physics data distributed over more than a hundred grid sites. Every day, about 5 million files [66] are transferred through the DDM system. DDM is not a stand-alone component of the distributed computing infrastructure. Due to the ATLAS computing model, failures in the DDM will cause PanDA jobs to fail and hence lead to a dramatic waste of computing power. In order to avoid massive failures, it is important to detect a decreasing data distribution efficiency in time. That is what the *DDM dashboard* [67] was designed for. As shown in figure 5.9, each site serves both as a source and as a destination for the experimental, processed, or simulated data. Therefore, it is important to gather the data transfer efficiency information for both theses roles separately.

In order to monitor the site performance in terms of DDM, the DDM Dashboard can be made use of, where information regarding grid jobs and transfers from different sources and locations is displayed. Besides a web interface, the DDM Dashboard provides monitoring data in a number of formats. Most suitable for parsing is the JSON format, due to its Python dictionary-like structure. Information about all space tokens of a specified site is extracted and displayed in an HTML table. The problems that may occur during data access are manifold. In general, errors may occur due to site-related problems, which serves both a source and destination role for file transfers. These errors have to be



	TRANSFER-	REGISTRATION-	STAGING-	CERN+	DE+	ES+	FR+	IT+	ND+	NL+	UK+	US+
TOTAL-	100 % 210 MB/s	100 %	100 % 633 kB/s	100 % 4 MB/s	100 % 34 MB/s	100 % 27 MB/s	99 % 14 MB/s	100 % 69 MB/s	100 % 831 kB/s	63 % 2 kB/s	100 % 40 MB/s	100 % 20 MB/s
ES IFAE+	100 % 111 kB/s	100 %	100 % 0 kB/s			100 % 38 kB/s	100 % 73 kB/s					
ES IFIC-LCG2+	99 % 134 MB/s	100 %	100 % 0 kB/s	100 % 4 kB/s	100 % 6 MB/s	100 % 21 MB/s	98 % 14 MB/s	100 % 58 MB/s	100 % 1 kB/s	0 % 0 kB/s	99 % 37 MB/s	
ES LIP-COIMBRA+	100 % 2 MB/s	100 %	100 % 0 kB/s			100 % 1 MB/s			100 % 829 kB/s			
ES NCG-INGRID-PT+	33 % 0 kB/s	100 %	100 % 0 kB/s			100 % 0 kB/s				0 % 0 kB/s		
ES PIC+	100 % 48 MB/s	100 %	100 % 633 kB/s	100 % 4 MB/s	100 % 5 MB/s	100 % 5 MB/s	100 % 236 kB/s	100 % 12 MB/s	100 % 1 kB/s	100 % 2 kB/s	100 % 3 MB/s	100 % 20 MB/s
ES UAM-LCG2+	100 % 25 MB/s	100 %	100 % 0 kB/s		100 % 24 MB/s	100 % 2 kB/s		50 % 139 kB/s			100 % 314 kB/s	100 % 224 kB/s

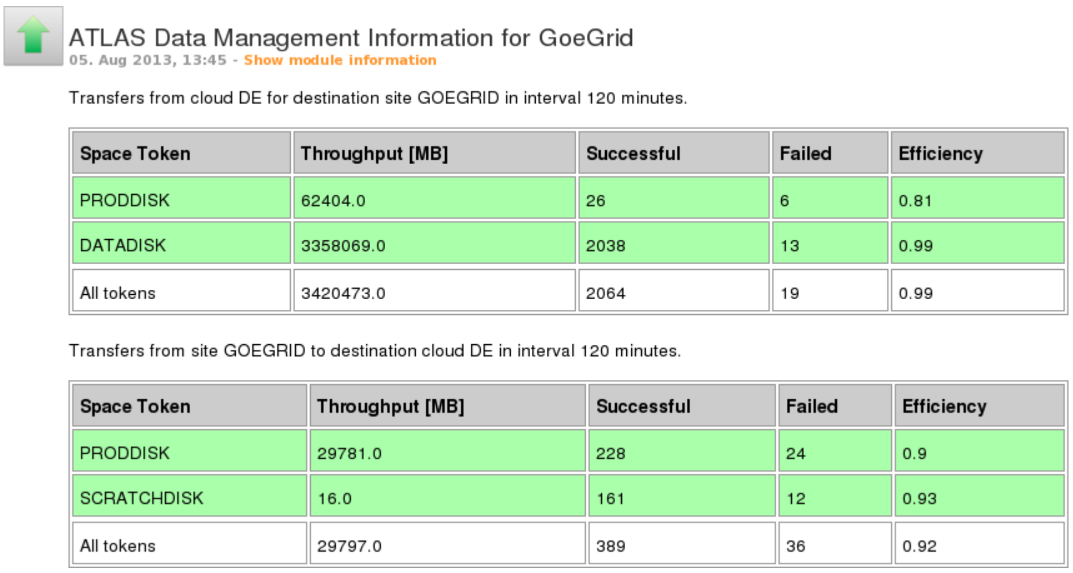
Figure 5.9.: ATLAS DDM dashboard web interface. Source and destination sites for data transfers are distinguished.

carefully distinguished and are displayed in separate tables. Finally, thresholds defined in the module configuration file determine the table row background colour and overall module status. The web output of this module is shown in figure 5.10.

5.9. DDM Deletion

The deletion of files in DDM is essential to clear disk space in grid sites of files no longer needed. This cleaning process is either initiated by a single user or centrally managed and executed by the individual grid sites. Due to inconsistencies between the central DDM database and the local storage management system, DDM deletion errors can occur. Because disk space is an expensive resource, these errors have to be monitored in order to ensure the cleaning process does not fail.

For the monitoring of the DDM deletion process, DDM provides a central web interface to all deletion process information, which can be queried for an output in JSON format. The obtained JSON file is downloaded and parsed. In an HTML table, the extracted data are separately displayed for datasets, files, and the overall volume. By the relative number of deletion errors, the data are assessed for the determination of the module status. All according thresholds are set in the module configuration file. In figure 5.11, the module HTML output is shown.



ATLAS Data Management Information for GoeGrid
05. Aug 2013, 13:45 - Show module information

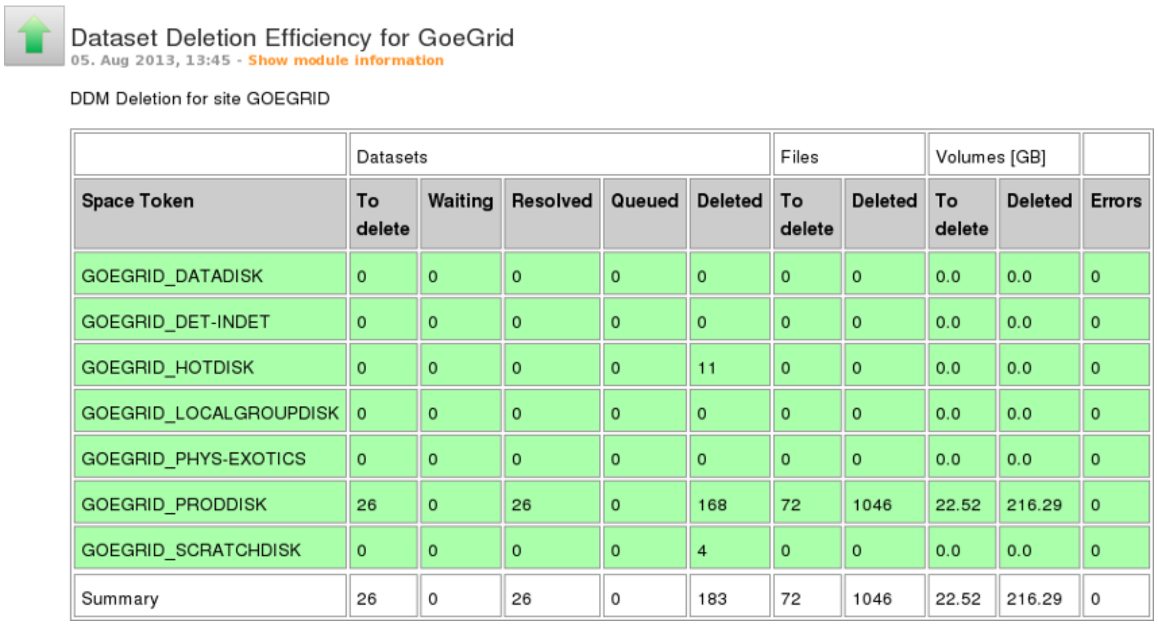
Transfers from cloud DE for destination site GOEGRID in interval 120 minutes.

Space Token	Throughput [MB]	Successful	Failed	Efficiency
PRODDISK	62404.0	26	6	0.81
DATADISK	3358069.0	2038	13	0.99
All tokens	3420473.0	2064	19	0.99

Transfers from site GOEGRID to destination cloud DE in interval 120 minutes.

Space Token	Throughput [MB]	Successful	Failed	Efficiency
PRODDISK	29781.0	228	24	0.9
SCRATCHDISK	16.0	161	12	0.93
All tokens	29797.0	389	36	0.92

Figure 5.10.: View of the DDM dashboard module. There are two separate tables for transfers with GoeGrid as destination and source of the transferred data.



Dataset Deletion Efficiency for GoeGrid
05. Aug 2013, 13:45 - Show module information

DDM Deletion for site GOEGRID

	Datasets					Files		Volumes [GB]		
Space Token	To delete	Waiting	Resolved	Queued	Deleted	To delete	Deleted	To delete	Deleted	Errors
GOEGRID_DATADISK	0	0	0	0	0	0	0	0.0	0.0	0
GOEGRID_DET-INDET	0	0	0	0	0	0	0	0.0	0.0	0
GOEGRID_HOTDISK	0	0	0	0	11	0	0	0.0	0.0	0
GOEGRID_LOCALGROUPDISK	0	0	0	0	0	0	0	0.0	0.0	0
GOEGRID_PHYS-EXOTICS	0	0	0	0	0	0	0	0.0	0.0	0
GOEGRID_PRODDISK	26	0	26	0	168	72	1046	22.52	216.29	0
GOEGRID_SCRATCHDISK	0	0	0	0	4	0	0	0.0	0.0	0
Summary	26	0	26	0	183	72	1046	22.52	216.29	0

Figure 5.11.: View of the DDM deletion module. The extracted data are separately displayed for datasets, files, and the overall volume.

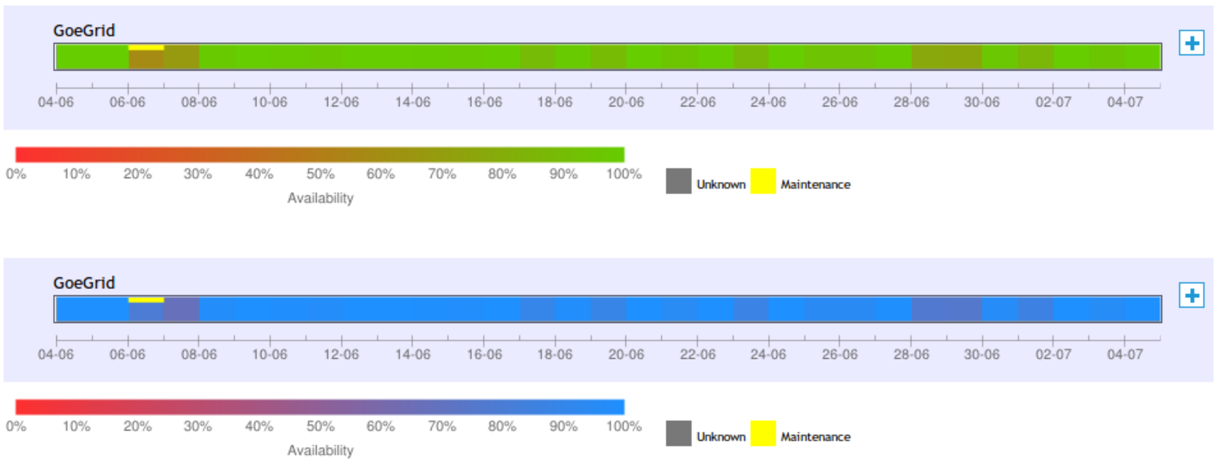
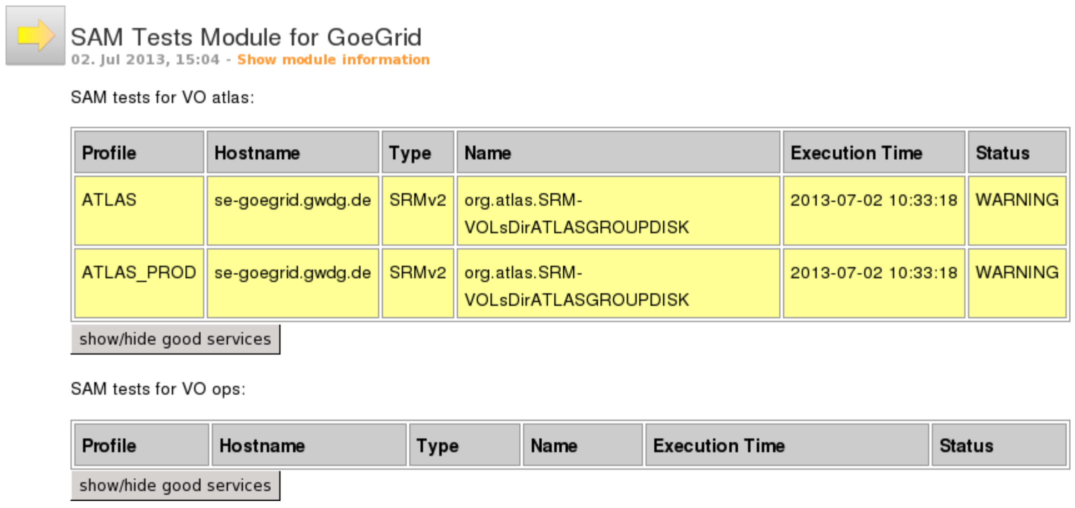


Figure 5.12.: Reliability and availability of GoeGrid from 2013-06-04 until 2013-07-04. The colour coding is defined by the MyWLCG SAM portal.

5.10. Service Availability Monitoring (SAM)

The grid sites from six VOs build the WLCG infrastructure, according to which resources are allocated. One of these organisations is *ATLAS*, providing computing resources for the ATLAS experiment. In order to ensure transparency in terms of grid job submission, all ATLAS sites also need to be a member of all according VOs, including *OPS* and *NGI-DE*. SAM monitors resources for the EGI, using certificates authorised by different VOs. The SAM tests monitor all kind of services that each site has to provide. Besides that, the site availability and reliability are determined. In figure 5.12, reliability and availability of GoeGrid from 2013-06-04 until 2013-07-04 are shown.

The SAM web page publishes the monitoring information in different formats, amongst others the JSON format. This module enables the simultaneous monitoring of SAM test results for multiple VOs, which can be specified in the module configuration file. A blacklist allows for the exclusion of certain services from monitoring. Furthermore, *warning* and *critical* thresholds are customisable. The extracted monitoring information is visualised in tables, one for each specified VO. For clarity and simplicity, only test results rated with a *warning* or *critical* status are displayed on the first glance. All other test results can be expanded. The web output of this module is shown in figure 5.13.


SAM Tests Module for GoeGrid
02. Jul 2013, 15:04 - Show module information

SAM tests for VO atlas:

Profile	Hostname	Type	Name	Execution Time	Status
ATLAS	se-goegrid.gwdg.de	SRMv2	org.atlas.SRM-VOLsDirATLASGROUPDISK	2013-07-02 10:33:18	WARNING
ATLAS_PROD	se-goegrid.gwdg.de	SRMv2	org.atlas.SRM-VOLsDirATLASGROUPDISK	2013-07-02 10:33:18	WARNING

SAM tests for VO ops:

Profile	Hostname	Type	Name	Execution Time	Status

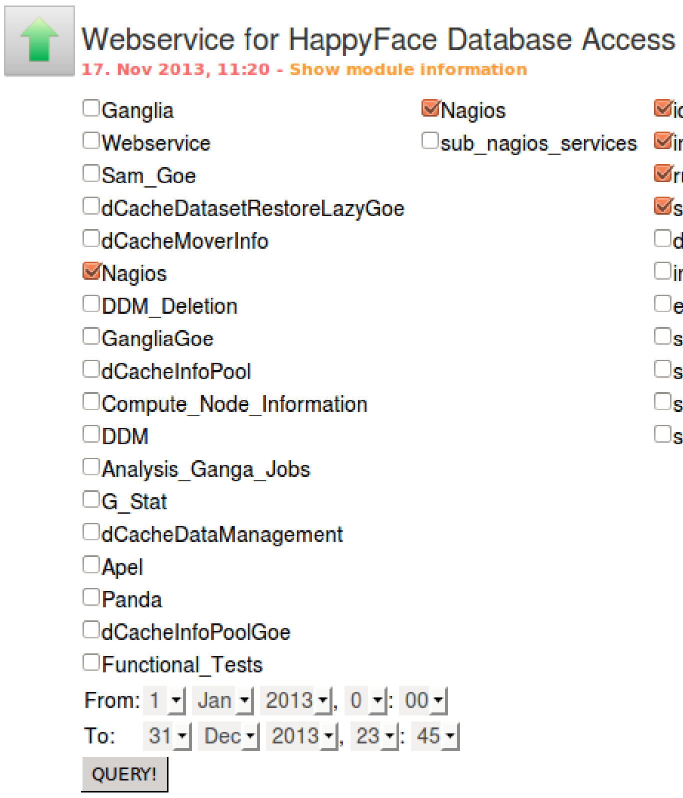
Figure 5.13.: HTML output of the SAM tests module. For the two VOs ATLAS and OPS, all monitored services are checked for their status. In the screenshot shown, two services exhibit a *warning* status and needed to be further investigated by the site administrators.

5.11. Web Service

The web service module is a rather atypical HappyFace module. It neither acquires any monitoring nor does it assess and present any monitoring data but is a user interface for the selection and retrieval of monitoring data previously acquired and stored by all other modules. Please note that being a GUI, the HappyFace web service module is not a web service.

The module Python source code passes the HappyFace database schema to the HTML template, which makes use of JavaScript to generate checkboxes for all modules in one column. Once a module is selected, its module table and all its subtables appear as checkboxes in a second column. When selecting one of the tables, all its table columns are visible in a third column and can be selected. Furthermore, a time interval for the requested monitoring data can be specified. By clicking the *Query!* button at the bottom of the module, the precedingly described RESTful web service is queried via a URL composed by the HappyFace module. Once the prepared file is ready for download, a download link to the query result becomes visible. In the downloadable file, all requested monitoring data are available in JSON format.

This module has proven very useful during the development of other HappyFace modules, as it provides a quick and easy-to-understand overview of the HappyFace database.


Webservice for HappyFace Database Access
17. Nov 2013, 11:20 - Show module information

☐ Ganglia	☑ Nagios	☑ id
☐ Webservice	☐ sub_nagios_services	☑ instance
☐ Sam_Goe		☑ run_id
☐ dCacheDatasetRestoreLazyGoe		☑ status
☐ dCacheMoverInfo		☐ description
☑ Nagios		☐ instruction
☐ DDM_Deletion		☐ error_string
☐ GangliaGoe		☐ source_url
☐ dCacheInfoPool		☐ services_ok
☐ Compute_Node_Information		☐ services_not_ok
☐ DDM		☐ services_critical
☐ Analysis_Ganga_Jobs		
☐ G_Stat		
☐ dCacheDataManagement		
☐ Apel		
☐ Panda		
☐ dCacheInfoPoolGoe		
☐ Functional_Tests		

From: 1 Jan 2013, 0 : 00
To: 31 Dec 2013, 23 : 45
QUERY!

Figure 5.14.: Web output of the web service module. This module basically is a GUI to the RESTful web service and provides a quick and easy-to-understand overview of the HappyFace database.

Furthermore, it served in the debugging of the RESTful web service and allows for a quick manual extraction of HappyFace monitoring data.

6. Conclusion and Outlook

6.1. Conclusion

Only the comprehensive monitoring of all hardware and software parts of a computing infrastructure guarantees its unobstructed operation with maximum performance. The HappyFace 3 meta-monitoring framework fulfils this task at the WLCG ATLAS Tier-2 centre GoeGrid and has proven its reliable operability during an extensive testing phase. During this period, both, the core of the HappyFace framework and all recently developed modules, were reviewed. The necessity of the further processing of previously gathered information required remote access to the HappyFace database. This entailed the development of a RESTful and a WSDL/SOAP-based web service, which allow to retrieve data from the HappyFace database regardless of the understanding of the HappyFace database structure.

The deployment of HappyFace version 3 as a replacement for the previous version 2 of the HappyFace meta-monitoring framework necessitated the redevelopment of all existing and used HappyFace modules. Newly emerging demands or thus far not considered grid site testing mechanisms also required the new development of a number of HappyFace modules. The modules currently in use in the HappyFace instance run at GoeGrid cover the full spectrum of local hardware, software, and service monitoring; central service and infrastructure monitoring; and the monitoring of results of external site testing mechanisms. Due to the wide use of generic monitoring systems in WLCG grid sites, such as Nagios and Ganglia, the modules developed for this purpose can be deployed to other grid sites, too. Furthermore, the ATLAS-wide use of the compute node information module, the analysis GANGA jobs module, the PanDA monitoring module, the APEL accounting module, the DDM dashboard monitoring module, the DDM deletion module, and the SAM module is easily possible. This flexibility required a generic design, easy configurability, and extensive testing. Several redeveloped modules constitute a considerable improvement and summary of modules that were previously split up in multiple parts. With the successful porting of old modules and the development of new modules for the GoeGrid HappyFace instance, the monitoring of GoeGrid as part of the WLCG infrastructure is ensured. All new modules especially provide the easy access to detailed information and the correlation of monitoring data.

Web services enable the communication of applications via a network, typically the internet. Thus, data can be retrieved and remote procedures can be called. For the remote access to data stored in the HappyFace database, two web services were developed. A lightweight and easy-to-use RESTful web services provides access the HappyFace database via HTTP GET requests. A standardised WSDL/SOAP-based web service provides a well-described interface to HappyFace. Furthermore, all stored data can also be selected via a web interface, which is integrated into HappyFace as a module. These interfaces allow the versatile retrieval of monitoring data that a HappyFace instance has aggregated. Well-structured return formats simplify the further processing of this data.

6.2. Outlook

HappyFace is not yet in wide use in WLCG ATLAS grid sites, though a wide use of this meta-monitoring tool is desirable for collaborative ATLAS-specific module development. In this context, HappyFace needs to be advertised throughout the ATLAS community. Until now, the installation and the setup of HappyFace requires a considerable amount of effort and therefore may not seem attractive to other ATLAS grid sites. The compilation and maintenance of an RPM package instead of an installation via subversion could convince sites not yet involved to use HappyFace.

Both, the RESTful and the WSDL/SOAP-based web service for access to the HappyFace database, do not require any authentication. In general, this is not security-critical since protected data can be chosen not to be offered by the web service server implementation. Nevertheless, the extension of both web services to inherit data protection from the individual HappyFace module configuration certificate authorisation settings would guarantee the smooth concurrence of HappyFace and the attached web services. In order to make the WSDL/SOAP-based web service a fully grid-enabled access method to HappyFace monitoring data, the use of *GSI*[1] would be favoured.

HappyFace is designed for meta-monitoring explicitly. Any kind of automatic adapta-

[1] *Grid Security Infrastructure* [68]

tion of warning thresholds or even action taking is not foreseen in the HappyFace framework and would require deep changes to the HappyFace core framework. As soon as root cause analysis and expert systems are available for GoeGrid, the integration of these systems into HappyFace allows new scopes for design. By the close collaboration of core and module developers, the integration of these future systems is possible and yields promising opportunities for the automation of cluster monitoring and maintenance.

A. Appendix

A.1. Installation of HappyFace 3 on CentOS 6.3

```
# obtain python-cherrypy3 from http://pkgs.org/centos-6-rhel-6/repoforge-i386/python-
    cherrypy-3.1.2-1.el6.rf.noarch.rpm.html
yum install ./python-cherrypy-3.1.2-1.el6.rf.noarch.rpm
yum install python-sqlalchemy
# obtain python-migrate from http://pkgs.org/centos-6-rhel-6/epel-i386/python-migrate
    -0.6-6.el6.noarch.rpm.html
yum install ./python-migrate-0.6-6.el6.noarch.rpm
yum install python-mako
yum install numpy
yum install python-matplotlib
svn co https://ekptrac.physik.uni-karlsruhe.de/public/HappyFace/branches/v3.0 HappyFace
cd HappyFace
svn co https://ekptrac.physik.uni-karlsruhe.de/public/HappyFaceModules/trunk modules
wget http://www-ekp.physik.uni-karlsruhe.de/~sroecker/files/hf3_config.tar.gz
tar -zxvf hf3_config.tar.gz
mv config/ HappyFace
yum install mod_wsgi
# copy Apache configuration to /etc/httpd/conf.d/
service httpd restart
```

A.2. Example Configuration for the HappyFace 3 Core

```
[paths]
happyface_url = /happy/

static_dir = static
archive_dir = %(static_dir)s/archive

tmp_dir = %(static_dir)s/tmp

hf_template_dir = templates
module_template_dir = modules/templates
template_cache_dir = mako_cache

template_icons_url = %(static_url)s/themes/armin_box_arrows

local_happyface_cfg_dir = config
```

```
category_cfg_dir = config/categories-enabled
module_cfg_dir = config/modules-enabled

acquire_logging_cfg = defaultconfig/acquire.log
render_logging_cfg = defaultconfig/render.log

# NOTE: Changing these URLs might have limited effect
static_url = %(happyface_url)sstatic
archive_url = %(static_url)s/archive


[happyface]
# colon separated list of categories, if empty
# all are used in a random order. The name here
# corresponds to the section name.
categories =
stale_data_threshold_minutes = 60

# automatic reload interval in minutes
reload_interval = 15

[auth]
# A file containing authorized DNs to access the site.
# One DN per line
dn_file =

# If the given DN is not found in the file above, if any, the following
# script is called with DN as first argument.
# The script must return 1 if user has access, 0 otherwise.
auth_script =

[template]
# relative to static URL
logo_img = /images/default_logo.jpg
documentation_url = https://ekptrac.physik.uni-karlsruhe.de/trac/HappyFace
web_title = HappyFace Project

[database]
url = sqlite:///HappyFace.db

[downloadService]
timeout = 300
global_options =

[plotgenerator]
enabled = False
backend = agg


[global]

server.socket_host: "0.0.0.0"
server.socket_port: 8080

tools.encode.on: True
tools.encode.encoding: "utf-8"
tools.decode.on: True
tools.trailing_slash.on: True
```

A.3. Example Apache Configuration for HappyFace 3

```
WSGIDaemonProcess hf_wsgi user=happy threads=20 processes=1
WSGISocketPrefix /var/run/wsgi
NameVirtualHost *:80

<VirtualHost *:80>
        Servername happyface3.local
        ServerAdmin christian@wehrberger.de
        DocumentRoot /home/happy/HappyFace
    #DocumentRoot /var/www/html/HappyFace
        WSGIProcessGroup hf_wsgi
        <Directory /home/happy/HappyFace>
                Order deny,allow
                Allow from all
        </Directory>
        WSGIScriptAlias /   /home/happy/HappyFace/render.py
        SetEnv configuration /home/happy/HappyFace
</VirtualHost>
```

A.4. A WSDL/SOAP-based Web Service for Access to the HappyFace Database in Python

A.4.1. WSDL Document

```
<?xml version="1.0" ?>
<definitions name="DatabaseService" targetNamespace="urn:ZSI" xmlns="http://schemas.
    xmlsoap.org/wsdl/" xmlns:myType="DatabaseTable_NS" xmlns:soap="http://schemas.xmlsoap
    .org/wsdl/soap/" xmlns:tns="urn:ZSI" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <schema targetNamespace="DatabaseTable_NS" xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="time_interval">
        <sequence>
          <element name="from" type="xsd:dateTime"/>
          <element name="to" type="xsd:dateTime"/>
        </sequence>
      </complexType>
      <complexType name="instance">
    <sequence>
      <element name="instance" type="xsd:string"/>
    </sequence>
      </complexType>
      <complexType name="response_mod_webservice">
        <sequence>
```

```
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.id" type="xsd:decimal"/>
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.instance" type="xsd:string"/>
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.run_id" type="xsd:decimal"/>
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.status" type="xsd:float"/>
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.description" type="xsd:string"/>
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.instruction" type="xsd:string"/>
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.error_string" type="xsd:string"/>
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.source_url" type="xsd:string"/>
     <element maxOccurs="unbounded" minOccurs="0" name="mod_webservice.database_structure_file" type="xsd:string"/>
    </sequence>
   </complexType>
   <complexType name="request_mod_webservice">
    <sequence>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.id" type="xsd:boolean"/>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.instance" type="xsd:boolean"/>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.run_id" type="xsd:boolean"/>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.status" type="xsd:boolean"/>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.description" type="xsd:boolean"/>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.instruction" type="xsd:boolean"/>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.error_string" type="xsd:boolean"/>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.source_url" type="xsd:boolean"/>
     <element maxOccurs="1" minOccurs="0" name="mod_webservice.database_structure_file" type="xsd:boolean"/>
    </sequence>
   </complexType>
   [...]
  </schema>
 </types>
 <message name="response_mod_webservice">
  <part name="mod_webservice" type="myType:response_mod_webservice"/>
 </message>
 <message name="request_mod_webservice">
  <part name="time_interval" type="myType:time_interval"/>
  <part name="mod_webservice" type="myType:request_mod_webservice"/>
  <part name="instance" type="myType:instance"/>
 </message>
 [...]
 <portType name="mod_webservice">
```

```
    <operation name="mod_webservice">
      <input message="request_mod_webservice"/>
      <output message="response_mod_webservice"/>
    </operation>
  </portType>
  [...]
  <binding name="mod_webservice" type="mod_webservice">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="mod_webservice">
      <soap:operation soapAction="urn:crafted.wsdl#mod_webservice"/>
      <input>
        <soap:body namespace="urn:crafted.wsdl" use="literal"/>
      </input>
      <output>
        <soap:body namespace="urn:crafted.wsdl" use="literal"/>
      </output>
    </operation>
  </binding>
  [...]
  <service name="Database Service">
    <documentation>Database Web Service</documentation>
    <port binding="tns:mod_webservice" name="mod_webservice">
      <soap:address location="http://localhost:7000/mod_webservice"/>
    </port>
    [...]
    </port>
  </service>
</definitions>
```

A.4.2. Server Implementation

```python
import time
from time import mktime
from datetime import datetime
from optparse import OptionParser
from ZSI.wstools import logging
from ZSI.ServiceContainer import AsServer
from DatabaseService_server import *
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

# Setup the database
def db_setup(connection_string='sqlite:///HappyFace.db', echo=False):
  engine = create_engine(connection_string, echo=echo)
  Session = sessionmaker(bind=engine, autoflush=False, autocommit=False)
  session = Session()
  metadata = MetaData(engine)
  return session, engine, metadata

session, engine, metadata = db_setup()
metadata.reflect()
metadata.tables.keys()
Base = declarative_base()
```

```

class table_hf_runs(Base):
  __table__ = Table('hf_runs', metadata, autoload=True)

class table_mod_webservice(Base):
  __table__ = Table('mod_webservice', metadata, autoload=True)

class MyDatabaseService_mod_webservice(Database_Service):
    _wsdl = "".join(open("crafted.wsdl").readlines())

    def soap_mod_webservice(self, ps, **kw):
        # Call the generated base class method to get appropriate
        # input/output data structures
        request, response = Database_Service.soap_mod_webservice(self, ps, **kw)

        time_from = datetime.fromtimestamp(mktime(request._time_interval._from))
        time_to = datetime.fromtimestamp(mktime(request._time_interval._to))

        class mod_webservice:
            pass

        for column in dir(request._mod_webservice):
            if str(column)[0] == '_' and str(column)[1] != '_':
                if getattr(request._mod_webservice, str(column)):
                    column_name = column[len('mod_webservice_')+1:]
                    if (not request._time_interval._from is None) and (not request._time_interval._to is None):
                        query = session.query(getattr(table_mod_webservice, column_name)).filter(table_hf_runs.id == table_mod_webservice.id).filter(table_hf_runs.time > time_from).filter(table_hf_runs.time < time_to).filter(table_mod_webservice.instance == request._instance._instance).all()
                    elif not request._time_interval._from is None:
                        query = session.query(getattr(table_mod_webservice, column_name)).filter(table_hf_runs.id == table_mod_webservice.id).filter(table_hf_runs.time > time_from).filter(table_mod_webservice.instance == request._instance._instance).all()
                    elif not request._time_interval._to is None:
                        query = session.query(getattr(table_mod_webservice, column_name)).filter(table_hf_runs.id == table_mod_webservice.id).filter(table_hf_runs.time < time_to).filter(table_mod_webservice.instance == request._instance._instance).all()
                    setattr(mod_webservice, str(column), [element[0] for element in query])
        response._mod_webservice = mod_webservice
        return request, response

op = OptionParser(usage="%prog [options]")
op.add_option("-l", "--loglevel", help="loglevel (DEBUG, WARN)",metavar="LOGLEVEL")
op.add_option("-p", "--port", help="HTTP port",metavar="PORT", default=7000, type="int")
options, args = op.parse_args()
# set the loglevel according to cmd line arg
if options.loglevel:
  loglevel = eval(options.loglevel, logging.__dict__)
  logger = logging.getLogger("")
  logger.setLevel(loglevel)
# Run the server with a given list services
print 'Started ...'
AsServer(port=options.port, services=[[...],MyDatabaseService_mod_webservice('
    mod_webservice')])
```

A.4.3. Client Implementation

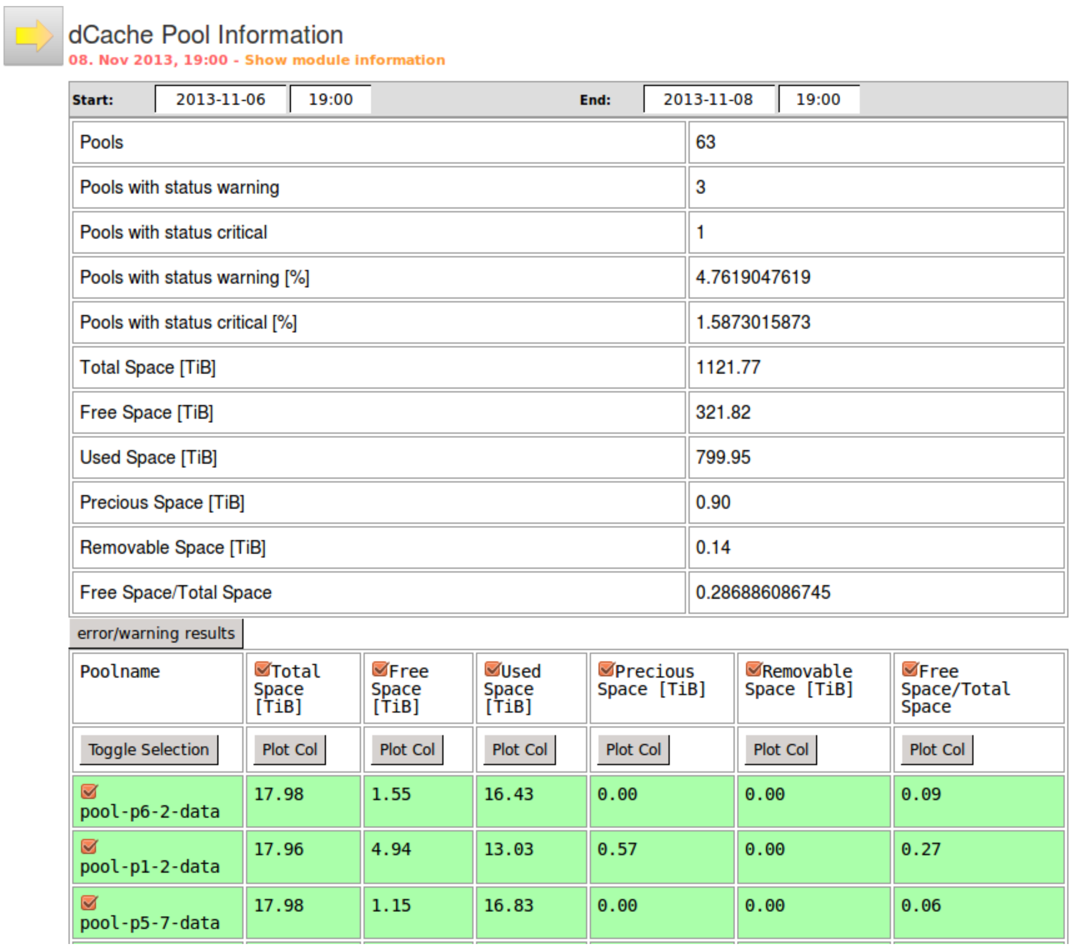
```
import sys, time
from optparse import OptionParser
from ZSI.version import Version as zsiversion
print "*** ZSI version %s ***" % (zsiversion,)
from DatabaseService_client import *
from inspect import *

def main():
    op = OptionParser(usage="%prog [options]")
    op.add_option("-u", "--url", help="service URL", metavar="URL")
    op.add_option("-i", "--input", type="string",
    help="input string for getCurrentDate WS method",metavar="INPUT")
    options, args = op.parse_args()
    loc = Database_ServiceLocator()

    service = loc.getmod_webservice(url=options.url, tracefile=sys.stdout)
    Request = request_mod_webservice()
    #print getmembers(Request.set_element_mod_webservice)
    class mod_webservice:
        _mod_webservice_id = True
    Request.set_element_mod_webservice(mod_webservice)
    class time_interval:
        _from = [2013, 1, 1, 0, 0, 3, 227, -1]
        _to = [2013, 8, 5, 0, 0, 3, 227, -1]
    Request.set_element_time_interval(time_interval)
    Response = service.mod_webservice(Request)
    print Response._mod_webservice._mod_webservice_id
    print Response._mod_webservice._mod_webservice_instance
    [...]

if __name__ == '__main__':
    main()
```

A.5. dCache Modules


dCache Pool Information

08. Nov 2013, 19:00 - Show module information

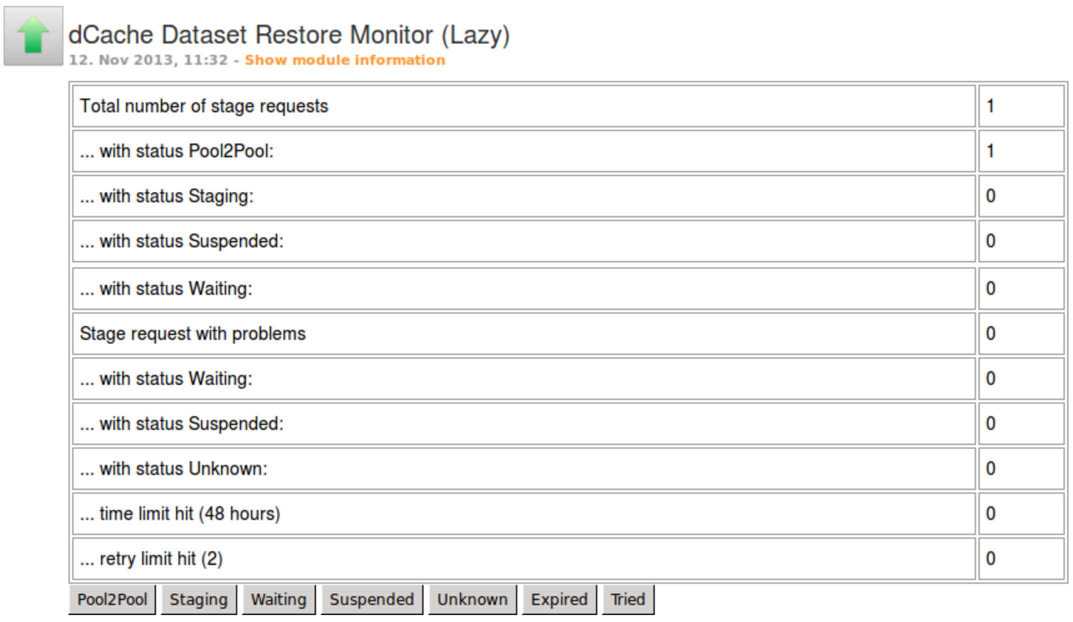
Start:	2013-11-06	19:00	End:	2013-11-08	19:00

Pools	63
Pools with status warning	3
Pools with status critical	1
Pools with status warning [%]	4.7619047619
Pools with status critical [%]	1.5873015873
Total Space [TiB]	1121.77
Free Space [TiB]	321.82
Used Space [TiB]	799.95
Precious Space [TiB]	0.90
Removable Space [TiB]	0.14
Free Space/Total Space	0.286886086745

error/warning results

Poolname	✓Total Space [TiB]	✓Free Space [TiB]	✓Used Space [TiB]	✓Precious Space [TiB]	✓Removable Space [TiB]	✓Free Space/Total Space
Toggle Selection	Plot Col	Plot Col	Plot Col	Plot Col	Plot Col	Plot Col
✓ pool-p6-2-data	17.98	1.55	16.43	0.00	0.00	0.09
✓ pool-p1-2-data	17.96	4.94	13.03	0.57	0.00	0.27
✓ pool-p5-7-data	17.98	1.15	16.83	0.00	0.00	0.06

Figure A.1.: Web output of the dCache pool information module. The parsing part of this dCache module had to be adapted to the dCache version run in GoeGrid. This module displays overall status and storage space information on all dCache pools and provides a plotting functionality using the HappyFace plot generator.


dCache Dataset Restore Monitor (Lazy)

12. Nov 2013, 11:32 - Show module information

Total number of stage requests	1
... with status Pool2Pool:	1
... with status Staging:	0
... with status Suspended:	0
... with status Waiting:	0
Stage request with problems	0
... with status Waiting:	0
... with status Suspended:	0
... with status Unknown:	0
... time limit hit (48 hours)	0
... retry limit hit (2)	0

Figure A.2.: Web output of the dCache dataset restore monitoring module. The parsing part of this dCache module had to be adapted to the dCache version run in GoeGrid. This module detects problems with dCache stage requests.

Bibliography

[1] D. C. van der Ster, J. Elmsheuser, M. Ú. García, M. Paladin, *HammerCloud: A Stress Testing System for Distributed Analysis*, in *Journal of Physics: Conference Series*, volume 331, 7, 072036, IOP Publishing (2011)

[2] T. Berners-Lee, R. Fielding, H. Frystyk, *Hypertext transfer protocol–HTTP/1.0* (1996)

[3] M. Delfino, L. Robertson, *Solving the LHC Computing Challenge: A Leading Application of High Throughput Computing Fabrics combined with Computational Grids*, CERN-IT-DLO-2001-003 (2001)

[4] ALICE Collaboration, *ALICE Technical Design Report*, CERN/LHCC 2001-021 (2001)

[5] ATLAS Collaboration, *ATLAS: technical proposal for a general-purpose pp experiment at the large hadron collider at CERN*, CERN/LHCC pages 171–173 (1994)

[6] CMS Collaboration, *CMS - Technical Proposal*, CERN/LHCC/94-38 (1994)

[7] LHCb Collaboration, *LHCb Technical Proposal*, CERN/LHCC 98-4 (1998)

[8] L. Evans, *The large hadron collider project*, CERN Reports pages 275–286 (1997)

[9] R. Jones, D. Barberis, *The ATLAS computing model*, in *Journal of Physics: Conference Series*, volume 119, 7, 072020, IOP Publishing (2008)

[10] T. Maeno, *PanDA: distributed production and distributed analysis system for ATLAS*, in *Journal of Physics: Conference Series*, volume 119, 6, 062036, IOP Publishing (2008)

[11] *ATLAS Distributed Data Management*, URL `https://twiki.cern.ch/twiki/bin/view/Atlas/DistributedDataManagement`

[12] V. Büge, V. Mauch, G. Quast, A. Scheurer, A. Trunov, *Site specific monitoring of multiple information systems - the HappyFace Project*, Journal of Physics **Conference Series 219, 062057** (2010)

[13] J. Meyer, A. Quadt, P. Weber, et al., *ATLAS Tier-2 at the Compute Resource Center GoeGrid in Göttingen*, in *Journal of Physics: Conference Series*, volume 331, 7, 072055, IOP Publishing (2011)

[14] *GWDG*, URL `http://www.gwdg.de/`

[15] *HEP-Grid*, URL `http://documentation.hepcg.org/`

[16] *Red Hat Enterprise Linux*, URL `http://www.redhat.com/products/enterprise-linux`

[17] *Scientific Linux CERN*, URL `http://linux.web.cern.ch/linux/scientific6`

[18] M. Burgess, R. Ralston, *Distributed resource administration using cfengine*, Software: practice and experience **27(9)**, 1083 (1997)

[19] F. D. Sacerdoti, S. Chandra, K. Bhatia, *Grid systems deployment & management using Rocks*, in *Cluster Computing, 2004 IEEE International Conference on*, pages 337–345, IEEE (2004)

[20] *Kernel-based Virtual Machine*, URL `http://www.linux-kvm.org/page/Main_Page/`

[21] *Autofs automatic file system mounter*, URL `http://www.autofs.org/`

[22] *dCap Protocol*, URL `http://www.dcache.org/manuals/Book-1.9.5/cookbook/cb-proto-dcap-passive.shtml`

[23] W. Allcock et al., *GridFTP Protocol Specification*, Global Grid Forum Recommendation GFD.20 (2003)

[24] W. Allcock, I. Foster, S. Tuecke, A. Chervenak, C. Kesselman, *Protocols and services for distributed data-intensive science*, in *AIP Conference Proceedings*, volume 583, page 161 (2001)

[25] M. de Riese, P. Fuhrmann, T. Mkrtchyan, M. Ernst, A. Kulyavtsev, V. Podstavkov, M. Radicke, N. Sharma, D. Litvintsev, T. Perelmutov, T. Hesselroth, G. Behrmann, T. Zangerl, P. Millar, O. Syngea, A. Petersen, *The dCache Book for 1.9.12-series* (2011), URL `http://www.dcache.org/manuals/Book-1.9.12/Book-a4.pdf`

[26] A. Shoshani, A. Sim, J. Gu, *Storage resource managers: Middleware components for grid storage*, in *NASA Conference Publication*, pages 209–224 (2002)

[27] M. Ernst, P. Fuhrmann, M. Gasthuber, T. Mkrtchyan, C. Waldman, *dCache, a distributed storage data caching system*, CHEP2001 Conference Notes pages 152–156 (2001)

[28] *European Middleware Initiative*, URL `http://www.eu-emi.eu`

[29] C. Aiftimiei, P. Andreetto, S. Bertocco, S. Fina, S. Ronco, A. Dorigo, A. Gianelle, M. Marzolla, M. Mazzucato, M. Sgaravatto, *Job submission and management through web services: the experience with the CREAM service*, Journal of Physics: Conference Series **119**, 062004 (2008)

[30] M. Göhner, C. Rückemann, *Accounting-Ansätze im Bereich des Grid-Computing*, D-Grid, Fachgebiete Monitoring, Accounting und Billing im D-Grid-Integrationsprojekt (2006)

[31] *Torque Batch System*, URL `http://www.clusterresources.com/products/torque-resource-manager.php`

[32] *Maui Batch Scheduler*, URL `http://www.clusterresources.com/products/maui-cluster-scheduler.php`

[33] *Berkeley Database Information Index (BDII)*, URL `https://twiki.cern.ch/twiki/bin/view/EGEE/BDII`

[34] *Nagios Open Source Monitoring*, URL `http://www.nagios.org/`

[35] M. L. Massie, B. N. Chun, D. E. Culler, *The ganglia distributed monitoring system: design, implementation, and experience*, Parallel Computing **30(7)**, 817 (2004)

[36] *Open Grid Forum Web Page*, URL `http://www.ggf.org/`

[37] I. Foster, C. Kesselman, *The grid: blueprint for a new computing infrastructure*, Morgan Kaufmann (2004)

[38] S. Zanikolas, R. Sakellariou, *A taxonomy of grid monitoring systems*, Future Generation Computer Systems **21(1)**, 163 (2005)

[39] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, R. Wolski, *A grid monitoring architecture*, Citeseer (2002)

[40] *Python Programming Language - Official Website*, URL `http://www.python.org`

[41] *SQLite Database Engine*, URL `http://www.sqlite.org`

[42] *The World-Wide Web Consortium (WC3): HTML*, URL `http://www.w3.org/html`

[43] *Cherrypy: A Python Minimalist Framework*, URL `http://www.cherrypy.org`

[44] *Apache Software Foundation*, URL `http://www.apache.org`

[45] *Asynchronous JavaScript and XML*, URL `http://www.w3schools.com/ajax/`

[46] *Mako Templates for Python*, URL `http://www.makotemplates.org`

[47] *SQLAlchemy*, URL `http://www.sqlalchemy.org`

[48] *PostgreSQL database*, URL `http://www.postgresql.org/`

[49] *Python WSGI adapter module for Apache*, URL `https://code.google.com/p/modwsgi`

[50] *Subversion Version Control System*, URL `http://subversion.apache.org`

[51] *Red hat Package Manager*, URL `http://www.rpm.org/`

[52] *Apache Software Foundation: Apache License, Version 2.0*, URL `http://www.apache.org/licenses/LICENSE-2.0`

[53] *GNU Wget*, URL `http://www.gnu.org/software/wget`

[54] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, A. Secret, *The world-wide web*, Communications of the ACM **37(8)**, 76 (1994)

[55] *The Unicode Consortium*, URL `http://www.unicode.org`

[56] *UTF-8 Character Encoding*, URL `https://tools.ietf.org/html/rfc3629`

[57] *W3C*, URL `http://www.w3.org/`

[58] E. Laure, F. Hemmer, F. Prelz, S. Beco, S. Fisher, M. Livny, L. Guy, M. Barroso, P. Buncic, P. Kunszt, et al., *Middleware for the next generation Grid infrastructure*, Computing in High Energy Physics and Nuclear Physics (CHEP 2004) (2004)

[59] *WSDL XML schema definition*, URL `http://schemas.xmlsoap.org/wsdl/`

[60] R. Salz, C. Blunck, *ZSI: The Zolera Soap Infrastructure*, Chapters publishing (2007)

[61] K. Holtman, A. Mutz, *Transparent content negotiation in HTTP*, Technical report, RFC 2295, March (1998)

[62] *PanDA Shift Guide*, URL `https://twiki.cern.ch/twiki/bin/view/PanDA/PandaShiftGuide`

[63] K. Harrison, W. Lavrijsen, P. Mato, A. Soroko, C. Tan, C. E. Tull, N. Brook, R. Jones, *GANGA: a user-Grid interface for Atlas and LHCb*, arXiv preprint cs/0306085 (2003)

[64] *Open Science Grid*, URL `https://www.opensciencegrid.org`

[65] *Nordic DataGrid Facility*, URL `http://www.ndgf.org`

[66] V. Garonne, G. A. Stewart, M. Lassnig, A. Molfetas, M. Barisits, T. Beermann, A. Nairz, L. Goossens, F. B. Megino, C. Serfon, et al., *The ATLAS Distributed Data Management project: Past and Future*, in *Journal of Physics: Conference Series*, volume 396, 3, 032045, IOP Publishing (2012)

[67] *ATLAS DDM Dashboard Web Interface*, URL `http://dashb-atlas-data.cern.ch/ddm2`

[68] S. Tuecke, *Grid security infrastructure (GSI) roadmap*, in *Grid Forum Security Working Group Draft* (2001)

Acknowledgements

Firstly, I like to take this opportunity to thank Prof. Dr. Arnulf Quadt, who gave me the opportunity to write this thesis and offered me a warm welcome in his research group. I would also like to show my gratitude to Dr. Kevin Kröninger for the time and effort he puts into this thesis as a referee.

I am indebted to Dr. Gen Kawamura, Erekle Magradze, Haykuhi Musheghyan, and Dr. Jordi Nadal for their support and advice indispensable for the successful completion of my Master's Thesis. They have made available their support in a number of ways.

Eigenständigkeitserklärung - Statement of Authorship

Erklärung nach §13(8) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Georg-August-Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.

Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Declaration according to §13(8) of the Examination Regulations for the Bachelor's degree and the Master's degree of the Georg-August-University of Göttingen:

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All verbatim extracts have been distinguished, and all sources of information have been specifically acknowledged.

Furthermore, this thesis has not been accepted in any previous application for a degree neither at this university nor at any other.

Göttingen, 13.12.2013

(Christian Georg Wehrberger)