GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Fakultät für Physik

Bachelor's Thesis

Entwicklung von HappyFace Modulen für Atlas

Update of HappyFace modules for Atlas

prepared by

Lino Oscar Gerlach

from Kreuztal

at the II. Physikalischen Institut

Thesis number:	II.Physik-UniGö-BSc-2015/12
Thesis period:	13th April 2015 until 20th July 2015
First referee:	Prof. Dr. Arnulf Quadt
Second referee:	Prof. Dr. Stanley Lai

Abstract

This bachelor thesis is about the update of existing modules for the software framework HappyFace. HappyFace acquires and displays data concerning the Atlas grid from several different websites and the grid itself. Changes in these websites make it necessary to find new ways of acquiring the data. As the structure of the data can be changed as well, new methods of extracting the relevant information are needed to be implemented. Additionally, the way the data are presented can be improved by adding graphs and tables or changing already existing ones.
During the working period, three different modules were edited. The first one was built from scratch for testing and learning purposes. The other two are called DDM and Panda. Both modules needed updates in acquiring, extracting and displaying the data.

Keywords: Physics, Bachelor Thesis, WLCG, Monitoring, HappyFace

Contents

1. Introduction

The *Large Hadron Collider* is currently the biggest and most powerful particle collider on earth [1]. The amount of data produced by observing collisions makes a worldwide system of computing and storage units necessary. This system is the *Worldwide LHC Computing Grid* [2]. There are several so called *monitoring tools* to observe the status of the Grid. In order to have a compact view of the interesting monitoring tools, HappyFace, a meta-monitoring tool, was invented. It is made in a modular way, so that flexibility is guaranteed. The main part of this bachelor thesis was updating some of the existing modules. In the following, the used terms and concepts are explained briefly.

1.1. LHC

The LHC (*Large Hadron Collider*) is a particle collider at CERN, the European Organization for Nuclear Research, near Geneva in a circular tunnel with a length of about 27 km laying about 100 m underground. It is currently the largest and most powerful of its kind and designed to collide protons at a centre of mass energy of 14 TeV with a Luminosity of $10^{34}\text{cm}^{-2}\text{s}^{-1}$ [1]. The protons circulate in so called bunches with 25 ns of distance in time. There are 2808 of these bunches with up to 10^{11} particles each in the collider simultaneously [3]. Superconducting magnets, which are cooled down to a temperature of 1.9 K by liquid Helium and create an 8.33 T magnetic field, keep the particles in their orbit.
Protons are produced by ionizing Hydrogen at *LINAC II.* Before they enter the actual LHC, the protons pass through different other accelerators. After the last one, the SPS (*Super Proton Synchrotron*), they have an energy of 450 GeV each. Not only protons, but also Lead ions can be collided at the LHC. However, the beam energy is only 2.8 TeV in that case [4]. There are several experiments located around the collider. The four most important of them are ATLAS, CMS, LHCb and ALICE. ALICE is for examining quark-gluon-plasma, which is an extremely hot and dense state, that existed shortly after the big bang. At LHCb, the CP violation and decays of hadrons that contain bottom or charm quarks, are investigated. CMS is the heaviest detector at LHC. The research there focuses

on properties of the Higgs boson, supersymmetric particles and possible sub-structures. The biggest detector at Lhc is called Atlas. Its purpose is to investigate the properties of the Higgs boson and supersymmetry as well.

1.2. Atlas

The Atlas detector (*A Toroidal Lhc ApparatuS* [3]) is the biggest detector at Cern. Its dimensions are 25 m in height and 44 m in length. The overall weight of the detector is approximately 7000 tonnes. It is built symmetrically around the beam axis and consists of several layers of detectors. These are (from inner to outer layer): Tracking detector, calorimeter and muon chamber. Around the innermost chamber, there is a thin, superconducting solenoid. Additional magnets are located azimuthally to the calorimeters.
In the inner detector, there is a 2 T magnetic field. High resolution semiconductor detectors (pixel and strip detectors) help reconstructing vertices and measuring momenta of particles. Around these, there are mounted so called *straw tubes*. The pixel detector has 80.4 million readout channels which is by far the highest amount of all subdetectors in Atlas [3]. All in all, this leads to a nearly continuous particle tracking with an uncertainty of 100 μm in beam direction and 10 - 15 μm perpendicular to it [5].
One can separate the calorimeters into electromagnetic and hadronic ones. The electromagnetic calorimeter uses liquid Argon to produce Bremsstrahlung and pair production when hit by a charged particle or a photon. The detection itself is done by measuring the ionisation of the Argon. This kind of calorimeter is called homogeneous since absorber and detector are one unit. The advantage of a homogeneous construction lies in the possibility to measure the whole energy of the particle without any loss. In the hadronic calorimeter, there are alternating layers of absorbing (iron) and scintillating material (plastic). Because the absorber material does not need to scintillate, one can chose material with a very short radiation length. This allows for a more compact way of building and therefore an improved spatial resolution.
Since the probability for a charged particle to react in the electromagnetic calorimeter is quickly decreasing with increasing mass, muons are very unlikely to be detected there. That is why additional muon chambers are mounted. They are located at the outermost of the detector because all other detectable particles will not reach that location. In Atlas, the muon spectrometer consists of a long barrel and two strong end cap magnets generating strong bending power. Three layers of high precision tracking chambers lead to an excellent muon momentum resolution [3].

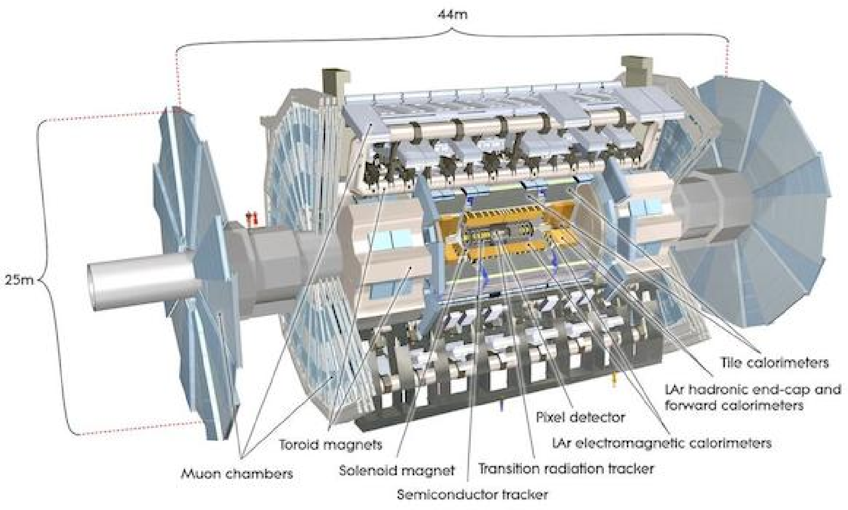

44m
25m
Tile calorimeters
LAr hadronic end-cap and forward calorimeters
Pixel detector
LAr electromagnetic calorimeters
Toroid magnets
Muon chambers
Solenoid magnet
Transition radiation tracker
Semiconductor tracker

Figure 1.1.: Cut-away view of the ATLAS detector [3].

1.3. WLCG

The amount of data that is produced at LHC is tremendous. 15 PetaBytes of data are expected to be generated every year, which is more than hundred times of what is produced at any other collider [2]. In addition to analysing measured events, computationally intensive simulations have to be done. This, and the fact that scientists all over the world need to have access to the data, propose the use of a worldwide grid. This grid is called WLCG (*Worldwide* LHC *Computing Grid*). In Fig.1.2, one can see the computing model for the ATLAS experiment. The main element of its infrastructure are the tiers 0 to 3 of computer centres:

- The Tier 0 centre at CERN accepts the data streaming from the experiment at full rates. It must also safe them to tape for archiving. A second copy of the whole data is stored among all Tier 1 centres. In order to make that data transfer possible, dedicated network connections to Tier 1 centres, with a bandwidth of 10 GBs^{-1} each, have been established. Together with all Tier 1 sites, Tier 0 is responsible for ensuring that data on tape are usable and accessible for the long term. It also provides a large computing cluster that comes into play when tasks need to take place close to the experiment. At the end of 2010, it contained some 50,000 CPU cores [6]. Finally, it is Tier 0's task to coordinate and overall operate the grid.

- There are thirteen Tier 1 sites of which one is in Canada, two in the United States, two in the Asia-Pacific region and eight in Europe [7]. Because they need to enable the usage and access of the data on tape for the long term, Tier 1 sites are usually

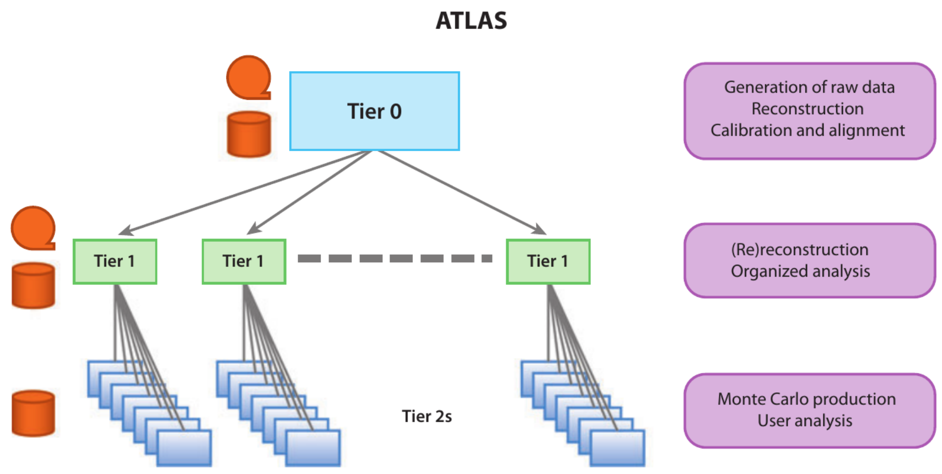


Figure 1.2.: ATLAS computing model [6].

large centres with experience in large-scale data management. Each site has to accept and store a previously agreed part of the raw and simulated data from Tier 0. There are computing clusters operated by Tier 1 sites that are used for processing the raw data with improved calibrations.

- Each Tier 1 site supports a number of Tier 2 centres. These are - in most cases - in the same country. Tier 2 sites do not only receive data from Tier 1, but also support in case of operational problems. They are essentially analysis facilities for the use of an entire experiment (in the US, they support only one experiment each, in Europe several ones or even all of them) and also provide computing resources for Monte Carlo simulations. The results of these simulations are sent to Tier 1 for archiving and processing. Tier 2 sites provide computing and storage resources, but do not necessarily archive data. This allows for small service teams running each Tier 2 site.

- Tier 3 sites receive datasets from Tier 2 sites and use them for end-user analysis. The scope of their hardware varies strongly, ranging from a few CPUs to large national analysis facilities [6]. Thus, they are not displayed in Figure 1.2.

1.4. GoeGrid

GoeGrid is a grid resource centre located in Göttingen, Germany. It is a joint project of high energy physics community, MediGrid, TextGrid, the department of physics of the University of Göttingen and the computer centre GWDG (*Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen*) which maintains the hardware, the network infrastructure and the water-cooling of the racks. GoeGrid's resources are used by a variety of scientific groups, which are also responsible for maintaining and financing. These groups do research in grid computing, software development, computer science, biomedicine, high energy physics, theoretical physics, and humanities [8]. Theoretical physicists use the computing power for quantum Monte Carlo simulations and numerical renormalisation group methods [9]. The high energy physics institute is part of the WLCG and serves as a Tier 2 and a Tier 3 centre for the ATLAS experiment.

1.4.1. Hardware setup

All in all, there were 2420 CPU-cores in the GoeGrid cluster by the end of 2010 [8]. This number has risen to 3668 by now (2016). In table 1.1, the different types of hardware used are listed. There are two different approaches to mass storage at GoeGrid. One is called Storage Area Network (SAN) and the other one, which is used by the high energy physics community, is called dCache. It is capable of storing and retrieving huge amounts of data, which are distributed among a large number of heterogenous server nodes, under a single virtual filesystem tree and offers a variety of standard access methods. 1 PB of disk space (upgraded from 516 TB in 2010) on local storage servers is managed by dCache for the high energy physics community. There are additional 30 TB of space available on the tape system of GWDG [8].

CPU	Cores	Clock [GHz]	Physical memory [GB]	Quantity
Intel X5355	8	2.66	16	78
Intel E5440	8	2.83	16	32
Intel E5530	16	2.4	24	1
Intel E5530	8	2.4	24	15
Intel X5650	12	2.66	37	56
Intel E5-2665	16	2.4	66	16
Intel E5-2660	16	2.2	66	16
Intel E5-2670	16	2.6	66	8
Intel E5-2660v2	20	2.2	132	13
Intel E5-2660v3	20	2.6	132	54

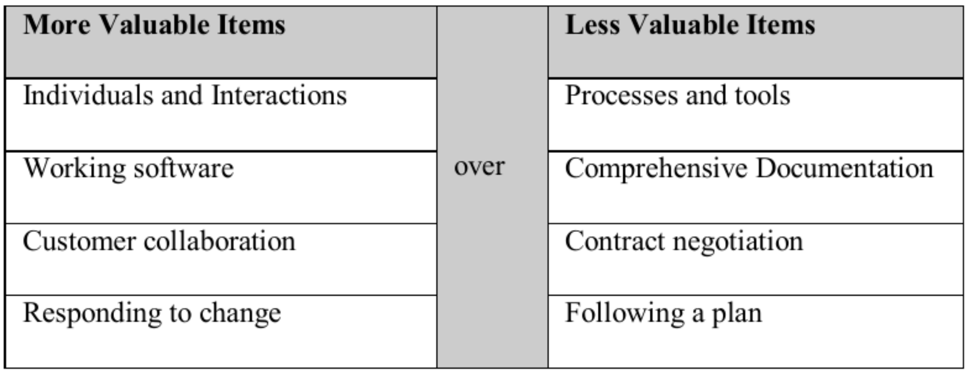
Table 1.1.: List of hardware types of the GoeGrid cluster (in 2016).

1.4.2. Software setup

All compute nodes have CentOS 6 Linux as their operating system. The users agreed to it, since it is compatible with all community specific criteria and applications and middlewares which is necessary for the interoperable use of the compute nodes. According to the requirements of the WLCG, the grid middleware gLite 3.1 (and 3.2) was chosen for the whole system. GoeGrid is accessible via the Globus Toolkit and gLite. Needed software packages' and updates' installation are managed by the cluster management tool Rocks. In this way, simple and fast changes of the configurations of all nodes are possible. The monitoring tool Ganglia is also included in Rocks [10].

1.5. Agile software development

There are several definitions of agile software development. They should all fulfill the so called *Agile Manifesto*. It was originally declared by K. Beck et al. in 2001 and states for example that a partly working software is more valuable than a comprehensive documentation, and that responding to current changes is more important than following the initial development plan. All four comparisons are shown in Fig.1.3. Note that software developers try to achieve the items on the right hand side nonetheless.



More Valuable Items		**Less Valuable Items**
Individuals and Interactions		Processes and tools
Working software	over	Comprehensive Documentation
Customer collaboration		Contract negotiation
Responding to change		Following a plan

Figure 1.3.: The Agile Manifesto [11].

The following definition of agile software development seems to be appropriate: *An iterative and incremental (evolutionary) approach to software development which is performed in a highly collaborative manner by self-organizing teams within an effective governance framework with "just enough" ceremony that produces high quality solutions in a cost effective and timely manner which meets the changing needs of its stakeholders* [11].
In case of dynamic software requirements (or requirements that are not completely known at the beginning of the project), the advantages of this software development approach

are obvious: Since the teams are self organised, less time is spent on arranging the different groups of programmers. The evolutionary progress of the software makes debugging easier because the last running version of the programme can be restored at any time. And in case the planned programme cannot be finished (i.e. because the time ran out), there is at least a latest version, which might fulfill a major part of the requirements. Most importantly, programmers can react to quickly changing goals.
Concerning HappyFace, software developers also use agile software development. The programme has a modular basis. One module is edited by one programmer at a time. The modified source codes are frequently uploaded to an online repository. Aftwerwards, they are run on a virtual test machine and subsequently deployed into the actual system.

2. The HappyFace project

Monitoring the status of the WLCG is a complex task. This is revealed by, among other things, investigating the workflow of the WLCG Transfers Dashboard. Fig.2.1 shows its initial architecture. FTS (*File Transfer System*) [12] is one of the two technologies used for data transfers. Monitoring it is a raw approximation of the global traffic between the four experiments' virtual organisations. AMQ stands for ActiveMQ which is a message broker that can also act as a buffer in front of the database [13].

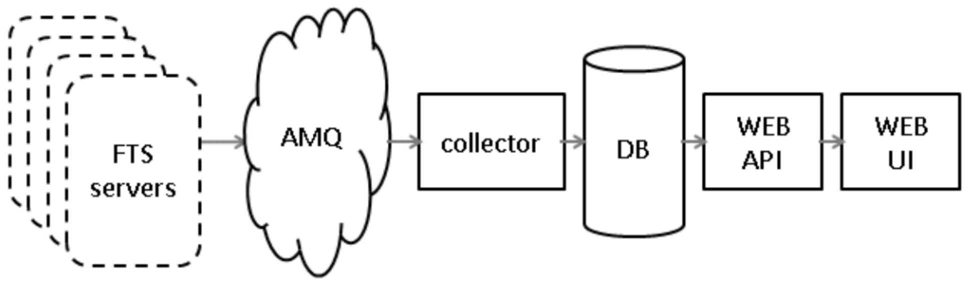


Figure 2.1.: Workflow of monitoring data transfers [13].

However, the WLCG Transfers Dashboard is just one of many monitoring tools. This causes some inconveniences. It is not only difficult to retrieve the needed information from a monitoring website as a person who does not do that frequently, but also the information on the status of the Grid is spread across numerous websites. Since most of the services that acquire and display this information were developed independently, the structure of the data and their presentation is not unified. This leads to difficulties for scientists who need to take information from several monitoring services into account. This multi monitoring is often necessary in order to identify fundamental problems that are only revealed by correlations between more shallow ones. Another handicap is the high latency of several monitoring websites. The databases might cause a delay between submitting the settings to a query and finally retrieving the relevant information. Scientists might spend an unnecessary amount of time for the regular site checks [14]. All these inconveniences when using several monitoring tools lead to the need for a so called *meta-monitoring* tool.

2.1. Meta-monitoring

In order to remedy the deficiencies described above, a system that acquires and displays relevant data from several monitoring websites is needed. It will not create any new monitoring information, but rather smartly summarise already existing ones. An automated workflow would save time and adaptable configurations make sure that changes to either the sources or the needs of the users can be dealt with. A good and user friendly meta-monitoring programme should also have the following features:

- **flexibility**: possibility to deploy the suite on virtually all common operating systems
- **aggregation**: all relevant information should be collected on one website
- **actuality**: the complete status information should be renewed with a regular time interval
- **traceability**: possibility to call the site status for a specific time respectively time scale
- **accessibility**: architecture should be as simple as possible for high performance availability
- **comfortability**: thought out navigation should realize a fast access to all results
- **alert functionality**: visualisation should highlight results via smileys, arrows, ...
- **customisability**: user should be able to implement easily tests and define alert algorithms

The list above can be found in [14]. HappyFace is a tool that has all these features.

2.2. HappyFace

HappyFace is a modular software framework. It was originally designed to query monitoring sources, to process the gathered information and to create an overview of the site status and its services [15]. Strictly speaking, it is, however, not a meta-monitoring tool anymore because it also aggregates information from the grid directly. The current version is HappyFace 3. In contrast to its predecessors, it clearly separates configuration and source code files and provides an automatically generated documentation [16].

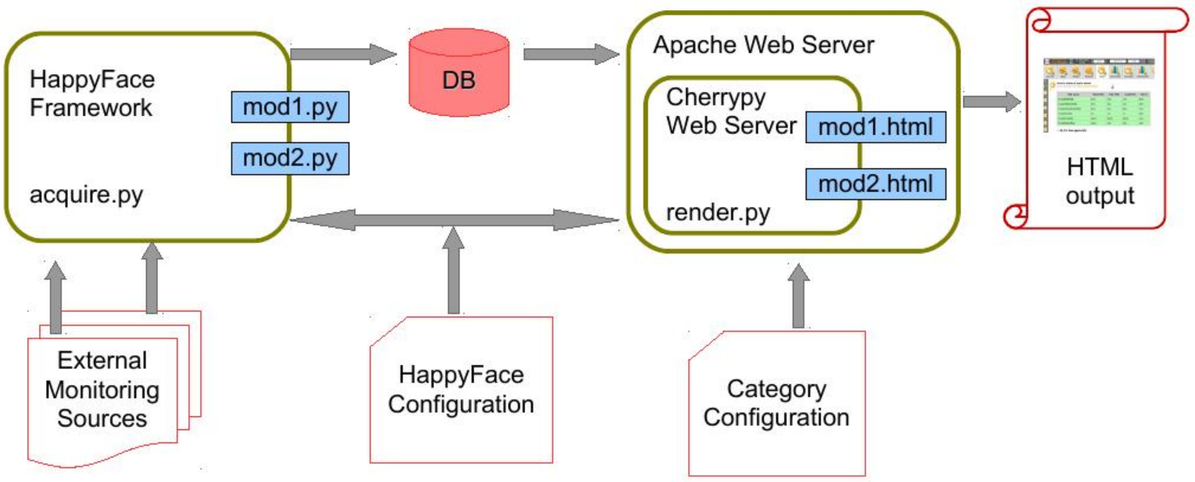


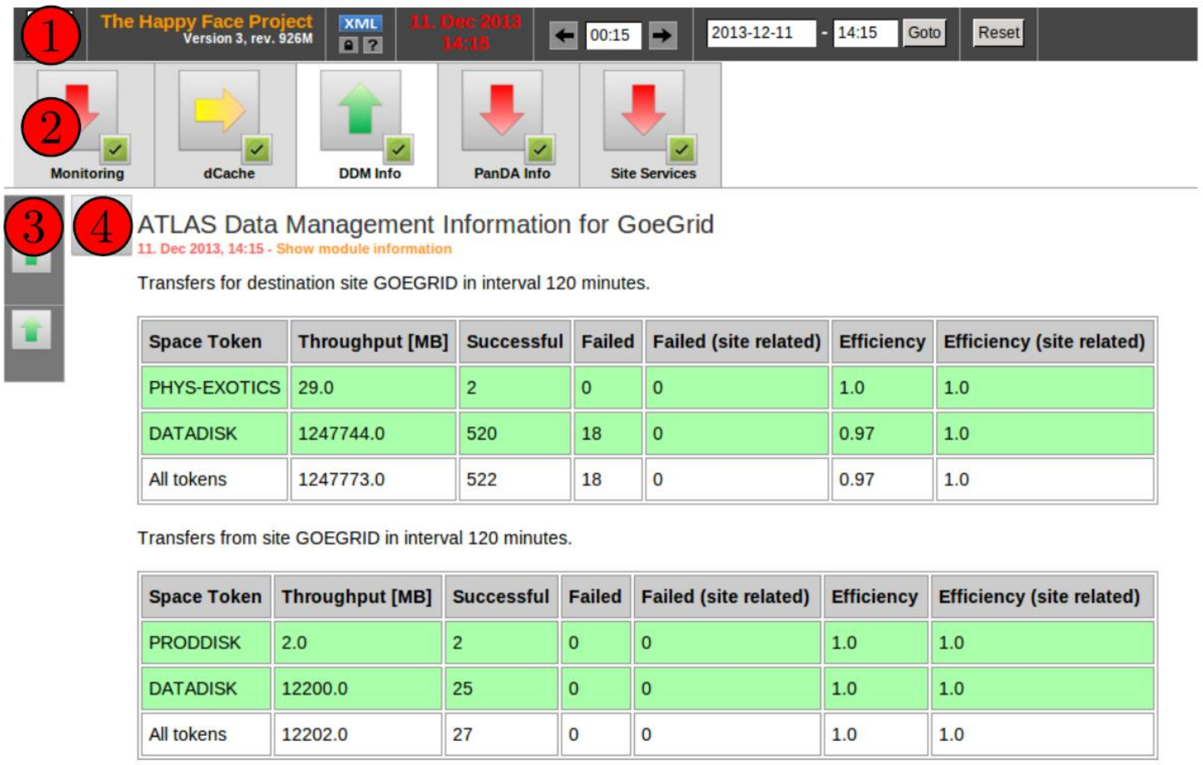
Figure 2.2.: Workflow of HappyFace [2].

2.2.1. Basic workflow

In order to grant maximal flexibility, acquiring and displaying the data need to take place separately. To make this possible, HappyFace makes use of an SQLite database, where all acquired data and any results derived from it are stored. Visualisation takes place with the help of a dynamic HTML output. This workflow also allows the traceability mentioned above since previously stored information can be accessed at any time. It is displayed in Fig.2.2.

The framework is split into two components: The core system called HappyCore and the system of modules (in Fig.2.2: `mod1` and `mod2`), which have to be programmed in a compulsory manner. Moreover, there must be config files for each module (HappyFace Configuration) and for each category of modules (Category Configuration), which define some basic parameters and determine whether the corresponding module is active or not. In addition to the python scripts `acquire.py` and `render.py`, which are responsible for executing all active modules and generating the website, the core system includes many necessary methods (i.e. the actual downloading of content) as well as optional ones (i.e. a plot generator). A screenshot of the HappyFace website can be seen in Fig.2.3.

The modular system offers various advantages. First of all, it enables agile software development since different programmers can work on different modules simultaneously and incremental changes to the software are possible. Deploying the edited modules happens quickly and in case of a failure, the former version can be restored quickly. It also makes the programme more customisable which is crucial because the sources might be changed and this can bring changes in the modules as well. Last but not least, the development of new modules can be simplified by inheriting from existing ones.



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 2.3.: Interface of HappyFace: title bar and history (1), category navigation bar (2), fast navigation bar (3) and individual module content (4) [16].

2.2.2. Rating system

For a quick overview of the status, an additional rating value is saved for each module. It is a float between 0 and 1, but can also be set to -1 or -2. Depending on this value, an icon for each category of modules is displayed on the HappyFace website. For values between 0 and 0.33, a red arrow pointing down, for values between 0.33 and 0.66, a horizontal yellow arrow and for values between 0.66 and 1, a green arrow pointing up is created. If the module execution fails for some reason, the value is set to -1 and if the data retrieval from the database fails, it is set to -2. For both negative values, the icon will contain a small exclamation mark. A summary of this rating system can be seen in Fig.2.4. In case there are multiple modules in one category, the lowest value is assumed for the whole category.

2.2.3. Implementation

The HappyCore is written in the script language Python. For each module, there are three files that can be edited to influence the outcome: a python script, an HTML file and a config file. For each category of modules, there is one config file.

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	$s_i = -2$

Figure 2.4.: HappyFace module rating schema [16].

The config files are read in by the Python ConfigParser. Each module can access configurations of all classes from which it is inherited. In case a variable is set multiple times by the configuration, the value that occurs last in the inheritance tree is chosen.
The data in the SQLite database mentioned above are encoded as ASCII. Binary files like plot images are stored in the local file system. Only their paths can be accessed via the database in order to keep it as fast as possible. There is one so called module table and an arbitrary amount of subtables for each module.
Depending on the time when the query was sent, the timestamp in the database closest to it is calculated and the corresponding information is displayed. This happens every five minutes per default.
To save time, all activated modules are executed at the same time in a multi-threading environment. As a result, the modules exceeding a global timeout setting can be terminated by the system. This way, in case an external source causes a delay, the system can continue the work [15].
In order to apply all changes made by developers to the system, the python script `acquire.py` has to be executed. It runs numerous essential functions from the HappyCore and the activated modules. This way, data from the monitoring sources is downloaded, processed and put in the database. It also partly refreshes the website. However, if changes to the *structure* of the database took place, its schema needs an additional update. This can be done by executing the python script `tools.py`. In case the HTML output for a module changed, the HTTP daemon needs to be restarted.

3. Module development

3.1. Basics

In the following section, the fictional module 'Mymodule' will be assumed. It is an empty module and contains only the needed parts of the source code in order not to produce an error message when the **acquire.py** script is running. The raw content of each of the three module specific files mentioned above will be explained briefly.

3.1.1. Mymodule.py

The python scripts of the modules are located under `/path/to/HappyFace3/modules/`. At the beginning of each script, the license and the editors have to be claimed. To do so, there is a predefined text to be commented out. The *Apache license*, a free license written by the Apache Software Foundation, is used for most modules. The rest of the script can be seen below in Listing 3.1. There are four functions that have to be implemented in each module's source code: `prepareAcquisition()` (*ll.* 15-18) for commissioning monitoring data downloads, `extractData()` (*ll.* 20-26) for the extraction of relevant information from the previously downloaded files, `fillSubtables()` (*ll.* 2-30) for the insertion of extracted data into module subtables and `getTemplateData()` (*ll.* 32-35) for creating variables that are accessible from the HTML mako template [16].
At first, functions from the HappyCore are imported (*l.* 1). These are saved in the `hf/` directory. In order to enable the use of the SQLite database, sqlalchemy has to be imported as well (*l.* 2). After that, the class of the module gets defined (*l.* 3). As one can see, it inherits from the `hf.module.ModuleBase`, which determines the basic structure of all modules. The `config_keys` are variables that will be read by the ConfigParser somewhere in the script (*ll.* 4-6). In case a variable is not mentioned in the configuration file, the `default_value` is assumed. Each module has one table called `table_columns`. For every column, the name and the type of the entry (TEXT or INT) have to be specified (*ll.* 7-9). Subtables can also be defined (*ll.* 10-13). Since there can be an arbitrary amount of them, each of them has a name. In this case it is just `subtable_name` (*l.* 11).

```
 1 import hf
 2 from sqlalchemy import *
 3 class Mymodule(hf.module.ModuleBase):
 4     config_keys = {
 5         'key': ('description', 'default_value')
 6     }
 7     table_columns = [
 8         Columns('column_name', TEXT)
 9     ], []
10     subtable_columns = {
11         'subtable_name': ([
12         Column('column_name', TEXT)
13     ], [])}
14 
15     def prepareAcquisition(self):
16         url = self.config['key']
17         self.source = hf.downloadService.addDownload(url)
18         self.subtable_name_db_value_list = []
19 
20     def extractData(self):
21         data = {
22             'column_name': 'entry'
23         }
24         content = open(self.source.getTmpPath()).read()
25         self.subtable_name_db_value_list=[{'column_name':'entry'}]
26     return data
27 
28     def fillSubtables(self, parent_id):
29         self.subtables['subtable_name'].insert().execute([dict(parent_id=parent_id,**row)
              for row in self.subtable_name_db_value_list])
30 
31     def getTemplateData(self):
32         data = hf.module.ModuleBase.getTemplateData(self)
33         details = self.subtables['subtable_name'].select().where(self.subtables['
             subtable_name'].c.parent_id==self.dataset['id']).execute().fetchall()
34         data['details'] = map(dict, details)
35         return data
```

Listing 3.1: Code extract from `Mymodule.py`.

At the beginning of the `prepareAcquisition` function, the URL to the file that should be downloaded is read from the configuration file (*l.* 16). After that, it is downloaded by the download service of the HappyCore, `hf.downloadService.addDownload(URL)`, and defined as `source` (*l.* 17). As a last step, the list of values to be entered into the subtables is defined (*l.* 18).

In the `extractData()` function, the already downloaded content is opened and read in (*l.* 24). After that, the actual preparation of the relevant data takes place. Since this process heavily depends on the details of each module, there is no code listed for this. At last, the columns of the main table receive their values (*ll.* 21-23). The same thing is done for the subtables (*l.* 25).

The `fillSubtables()` function is responsible for inserting all values into the corresponding subtables. This is done by an SQL command within a `for` loop (*l.* 29).
Finally, `getTemplateData()` defines `data` and `details` as lists that can then be accessed via the HTML file of the module (*ll.* 33-34). This happens with the help of a function from the HappyCore, `hf.module.ModuleBase.getTemplateData()`, and another SQL command (*l.* 34).

3.1.2. Mymodule.html

The HappyFace framework sets some regulations to the HTML files of the modules as well. They are so called mako templates. The templates also have to be saved to `/path/to/HappyFace3/modules/`. The content of the file `Mymodule.html` is displayed in Listing 3.2.
First of all, the character encoding is defined (*l.* 1). The most common character encoding in the world wide web is *UTF-8* [16]. Not only the python scripts, but also the HTML templates inherit from the module base (*l.* 2). Then, the actual content of the template file is introduced (*l.* 4) (and closed afterwards (*l.* 10)).
After that, text can be written in order to be displayed on the HappyFace website. In this case it simply states *Hello, World!* (*l.* 5). Access to the database is indicated by the dollar sign (*l.* 6). To display the value of the variable `column_name`, the line `$module.dataset['column_name']` has to be used. However, this way only works for variables saved in the module table. Variables from the subtables have to be accessed in a `for` loop. They can be implemented in the HTML template by using the percentage sign. It is an escape sequence and allows the embedding of python code (*ll.* 7-9). This technique allows variables with the same name in an arbitrary amount of subtables that can be queried via `for` loops. The module can create the proper amount of subtables and define all variables without the programmer knowing beforehand how many will be needed. Since the value of both variables is 'entry', the output on the HappyFace website would be *Hello, World!entryentry*

```
1  ## -*- coding: utf-8 -*-
2  <%inherit file="/module_base.html" />
3
4  <%def name="content()">
5  Hello, World!
6  ${module.dataset['column_name']}
7  % for detail in subtable_name:
8      ${detail['column_name']}
9  % endfor
10 </%def>
```

Listing 3.2: `Mymodule.html`.

3.1.3. mymodule.cfg

The config file for each active module is located under `/path/to/HappyFace3/config/modules-enabled/`. The content of the file `mymodule.cfg` is displayed in Listing 3.3.
At first, the corresponding module, its name and a short description have to be specified (*ll.* 1-4). The two parameters `type` and `weight` have influence on the way the module is presented on the HappyFace website. Most importantly, the config file must contain every `config_key` that is called in the `Mymodule.py` script. In this case, the only one is called `key` and should define the URL to the website from where the monitoring data are downloaded (*l.* 8).

```
1 [mymodule]
2 module = Mymodule
3 name = My empty module
4 description = empty module for demonstration purposes
5 type = rated
6 weight = 1.0
7
8 key = url_to_monitoring_data_source
```

Listing 3.3: `mymodule.cfg`.

3.2. JSON files

Extracting data that are included into an HTML file has several disadvantages. First of all, big amounts of unnecessary data are downloaded. Not only is this slowing down the complete working process of HappyFace, but also the resources of the website offering the data are stressed. This is especially the case because HappyFace will perform the downloads frequently. Another disadvantage of downloading HTML files lays in the fact that the layout or design of the queried website might be changed sometime. This will inevitably affect the HTML files as well. The algorithms responsible for extracting the relevant data from the HTML files might then need to be changed, too. A good data format for HappyFace to download needs to be independent of the design of the website. It should also be as compact as possible in order to reduce data traffic.
The most favourable format of monitoring data is JSON (*JavaScript Object Notation*). It was developed for the purpose of being human readable and easy for computers to parse and use. Although it is (as the name suggests) directly supported inside JavaScript and best suited for JavaScript applications, there are JSON parsers in every common programming language. JSON provides significant performance gains over other comparable formats. It is estimated to parse up to one hundred times faster than XML in modern

browsers [17]. In Listing 3.4 and 3.5, two different examples of files with the same content are displayed. The first one is an XML file, the other one a JSON file. They both contain the same list of two names split up into first and last name and are formatted to grant a good readability. Each element of a JSON file is surrounded by curly brackets and can contain elements with multiple elements in it and so on. When downloaded by HappyFace, JSON files usually do not contain any spacebars or new lines.

```
<name>
        <first>Cem</first>
        <last>Anhan</last>
</name>
<name>
        <first>Aykut</first>
        <last>Anhan</last>
</name>
```

Listing 3.4: `names.xml`.

```
{
        "firstname":"Cem",
        "lastname":"Anhan"
},
{
        "firstname":"Aykut",
        "lastname": "Anhan"
}
```

Listing 3.5: `names.json`.

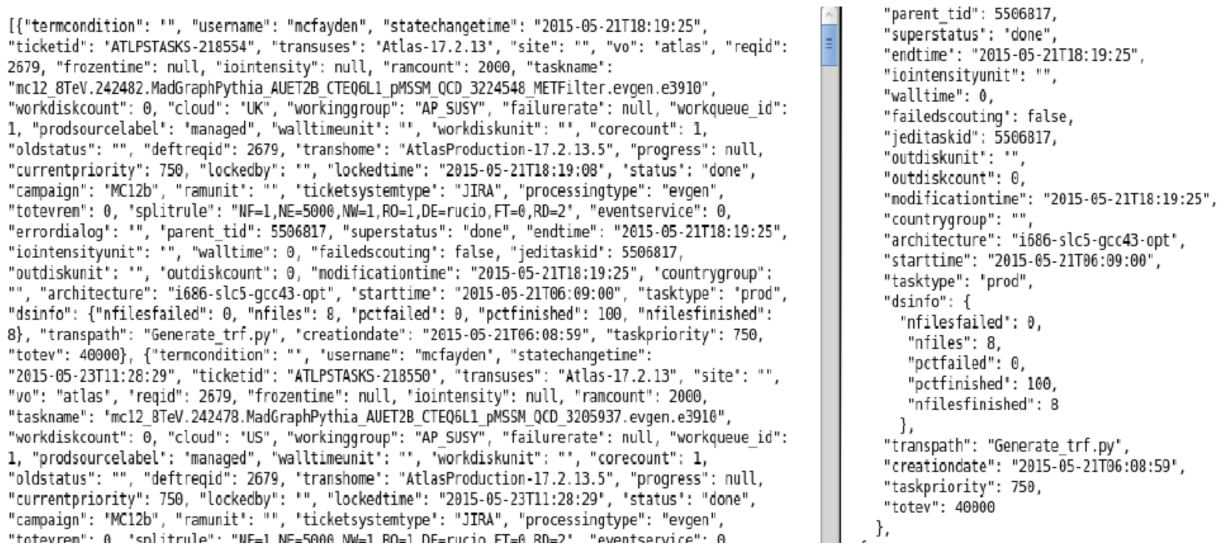
When downloading JSON files from monitoring websites such as BigPanDA (see chapters below), additional filtering parameters can be appended to the URL after a question mark. Assuming the URL to `names.json` was `http://www.bigpanda.cern.ch/names.json`, only the first half of the file would appear if the browser was given `http://www.bigpanda.cern.ch/names.json?firstname=Cem`. However, if `?lastname=Anhan` was appended instead, the output would remain untouched. Different parameters can be combined by using the & sign between them.
Since I had to read and understand many JSON files, an easily readable format is very helpful. For that purpose, I created a small programme. It is written in Python and formats JSON files by putting spacebars and new lines at suggestive points. This is achieved by counting the amount of opened curly brackets and putting an according amount of spacebars before each entry. Its code can be seen in Listing 3.6 and will be explained in the following.
At first, `sys` is imported (*l.* 1). This is necessary for the opportunity to quit the programme manually. At the beginning of the actual class, the JSON file that should be formatted is read in and defined as `destination` (*l.* 4). There is also a prompt to guide

the user which states “Name of the .json file to format: ”. The programme then tries to open the given file (*ll.* 4-5). In case the file does not exist or is not readable for other reasons, ”Not a valid file.” will be the output of the terminal and the programme is quit (*ll.* 7-9).

`fout` defines an output stream (*l.* 11). `n` is the variable that will count the number of needed spacebars later on. At the beginning, it is set to 0 (*l.* 12). The rest of the code is inside a `for` loop that runs through every single character of the file (*l.* 13). In case the character is a closed curly bracket, a new line and 3`n` spacebars are added and `n` is decreased by one (*ll.* 14-20). After that, the character itself is printed. The same thing happens if the character is an opened curly bracket except for the fact that `n` is increased by one instead (*ll.* 21-27). If the character is a comma, spacebars and new lines are added in the same way, but `n` remains untouched (*ll.* 27-31). At the end, the stream is closed again (*l.* 32).

```
1  import sys
2  
3  class Format:
4          destination = raw_input("Name␣of␣the␣.json␣file␣to␣format:␣")
5          try:
6                  content = open(destination).read()
7          except:
8                  print('Not␣a␣valid␣file.')
9                  sys.exit(0)
10 
11         fout = open(destination+'.txt', 'w')
12         n = 0
13         for i in range(len(content)):
14                 if content[i]=="}":
15                         fout.write('\n')
16                         j = 0
17                         for j in range(n):
18                                 fout.write('␣␣␣')
19                         n -= 1
20                 fout.write(content[i])
21                 if content[i]=="{":
22                         fout.write('\n')
23                         n += 1
24                         j = 0
25                         for j in range(n):
26                                 fout.write('␣␣␣')
27                 if content[i]==",":
28                         fout.write('\n')
29                         j = 0
30                         for j in range(n):
31                                 fout.write('␣␣␣')
32         fout.close()
```

Listing 3.6: format.py.



```
[{"termcondition": "", "username": "mcfayden", "statechangetime": "2015-05-21T18:19:25",
"ticketid": "ATLPSTASKS-218554", "transuses": "Atlas-17.2.13", "site": "", "vo": "atlas", "reqid":
2679, "frozentime": null, "iointensity": null, "ramcount": 2000, "taskname":
"mc12_8TeV.242482.MadGraphPythia_AUET2B_CTEQ6L1_pMSSM_QCD_3224548_METFilter.evgen.e3910",
"workdiskcount": 0, "cloud": "UK", "workinggroup": "AP_SUSY", "failurerate": null, "workqueue_id":
1, "prodsourcelabel": "managed", "walltimeunit": "", "workdiskunit": "", "corecount": 1,
"oldstatus": "", "deftreqid": 2679, "transhome": "AtlasProduction-17.2.13.5", "progress": null,
"currentpriority": 750, "lockedby": "", "lockedtime": "2015-05-21T18:19:08", "status": "done",
"campaign": "MC12b", "ramunit": "", "ticketsystemtype": "JIRA", "processingtype": "evgen",
"totevrem": 0, "splitrule": "NF=1,NE=5000,NW=1,RO=1,DE=rucio,FT=0,RD=2", "eventservice": 0,
"errordialog": "", "parent_tid": 5506817, "superstatus": "done", "endtime": "2015-05-21T18:19:25",
"iointensityunit": "", "walltime": 0, "failedscouting": false, "jeditaskid": 5506817,
"outdiskunit": "", "outdiskcount": 0, "modificationtime": "2015-05-21T18:19:25", "countrygroup":
"", "architecture": "i686-slc5-gcc43-opt", "starttime": "2015-05-21T06:09:00", "tasktype": "prod",
"dsinfo": {"nfilesfailed": 0, "nfiles": 8, "pctfailed": 0, "pctfinished": 100, "nfilesfinished":
8}, "transpath": "Generate_trf.py", "creationdate": "2015-05-21T06:08:59", "taskpriority": 750,
"totev": 40000}, {"termcondition": "", "username": "mcfayden", "statechangetime":
"2015-05-23T11:28:29", "ticketid": "ATLPSTASKS-218550", "transuses": "Atlas-17.2.13", "site": "",
"vo": "atlas", "reqid": 2679, "frozentime": null, "iointensity": null, "ramcount": 2000,
"taskname": "mc12_8TeV.242478.MadGraphPythia_AUET2B_CTEQ6L1_pMSSM_QCD_3205937.evgen.e3910",
"workdiskcount": 0, "cloud": "US", "workinggroup": "AP_SUSY", "failurerate": null, "workqueue_id":
1, "prodsourcelabel": "managed", "walltimeunit": "", "workdiskunit": "", "corecount": 1,
"oldstatus": "", "deftreqid": 2679, "transhome": "AtlasProduction-17.2.13.5", "progress": null,
"currentpriority": 750, "lockedby": "", "lockedtime": "2015-05-23T11:28:29", "status": "done",
"campaign": "MC12b", "ramunit": "", "ticketsystemtype": "JIRA", "processingtype": "evgen",
"totevrem": 0, "splitrule": "NF=1,NE=5000,NW=1,RO=1,DE=rucio,FT=0,RD=2", "eventservice": 0
```

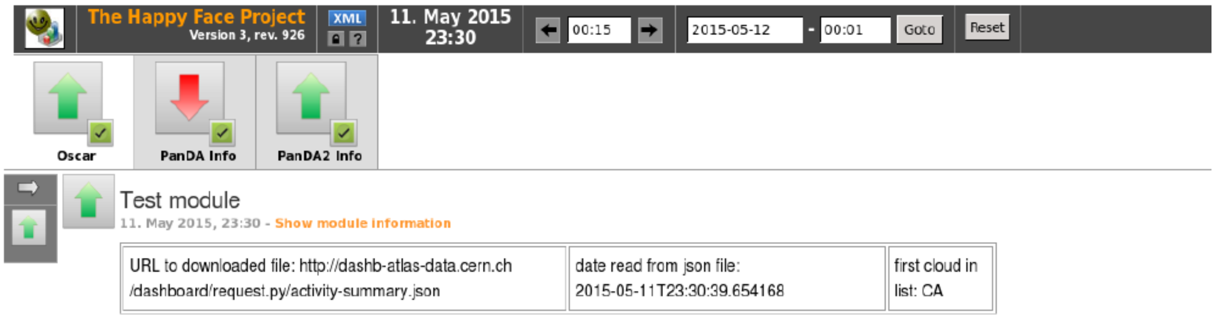
```
    "parent_tid": 5506817,
    "superstatus": "done",
    "endtime": "2015-05-21T18:19:25",
    "iointensityunit": "",
    "walltime": 0,
    "failedscouting": false,
    "jeditaskid": 5506817,
    "outdiskunit": "",
    "outdiskcount": 0,
    "modificationtime": "2015-05-21T18:19:25",
    "countrygroup": "",
    "architecture": "i686-slc5-gcc43-opt",
    "starttime": "2015-05-21T06:09:00",
    "tasktype": "prod",
    "dsinfo": {
      "nfilesfailed": 0,
       "nfiles": 8,
       "pctfailed": 0,
       "pctfinished": 100,
       "nfilesfinished": 8
      },
    "transpath": "Generate_trf.py",
    "creationdate": "2015-05-21T06:08:59",
    "taskpriority": 750,
    "totev": 40000
  },
```

Figure 3.1.: Original JSON file (left) and formatted one (right).

Such a functionality did already exist. Unfortunately, I was informed about it after finishing the programme. It can be used by simply specifying *prettify* as one of the parameters in the URL. However, in Fig.3.1, a json file is displayed before and after formatting with my programme.

3.3. Test module

In order to understand how module development for HappyFace works, I created a simple test module called Oscar. It has a similar structure to Mymodule, which was described in a previous chapter. The output of this module can be seen in Fig.3.2. There are three columns in the module's database. It downloads a JSON file from the old DDM website called activity-summary and reads in the so called `toDate`. The downloaded JSON files only cover a certain time span. `toDate` is the point in time when this time span ended. By the way, the beginning is called `fromDate`. The second information that is extracted from the JSON file is the first name in the list of clouds. The last column of the data base simply contains the URL to the downloaded file. The content of the three columns is displayed in a table on the HappyFace website. All in all, the module is not very helpful. But as mentioned above, its sole purpose was getting to know the module development of HappyFace.
Although the test module is quiet simple, it took about three weeks until it was working. This time span includes installing HappyFace on pcatlas44, reading papers and correcting many mistakes with the help of the grid team.



URL to downloaded file: http://dashb-atlas-data.cern.ch/dashboard/request.py/activity-summary.json | date read from json file: 2015-05-11T23:30:39.654168 | first cloud in list: CA

Figure 3.2.: Output of the test module Oscar.

3.4. DDM

3.4.1. Preconditions

The ATLAS Distributed Data Management (DDM) is responsible for managing 90 PB of data that are spread across the WLCG. These consist of more than 8 PB of RAW data that were produced by the ATLAS detector since 2008 and many more derived products such as complementary simulation data [18]. The data have to be stored, moved and accessed by scientists. The system of choice to do that used to be Don Quijote 2 (DQ2). For the Run 2, it got replaced by its successor Rucio.
All the data produced by ATLAS are physically stored in files. They usually, but not exclusively, consist of persistent C++ objects associated with physics events. These files are the elemental objects handled by DDM. However, physicists are in most cases interested in complete sets of files that correspond to one topic or property. For this purpose, DDM allows the aggregation of files into so called *datasets*. They are the operational unit of DDM and can be transferred to different grid sites, whereas single files may not. Some files are in different datasets at the same time; so they may overlap. Several datasets are then collected in a so called *container*. They allow large blocks of data to be described in the system. A container typically contains similar datasets. These might for example originate from Monte Carlo simulations with similar parameters.
The task of DDM, managing the data, includes the following responsibilities [18]:

- Registering and cataloging ATLAS data
  - Registration of datasets and containers
  - Registration of files into datasets
  - Registration of datasets into containers
- Transferring data between sites
  - Registering the data transfer request

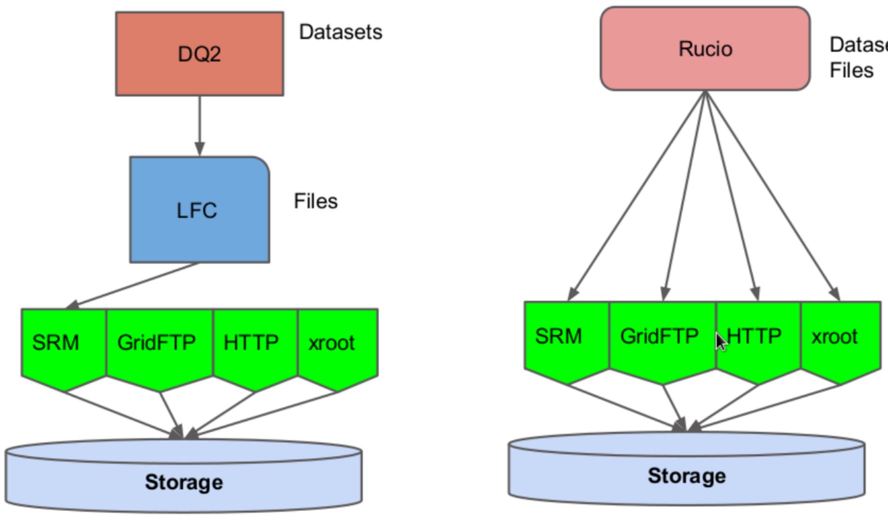
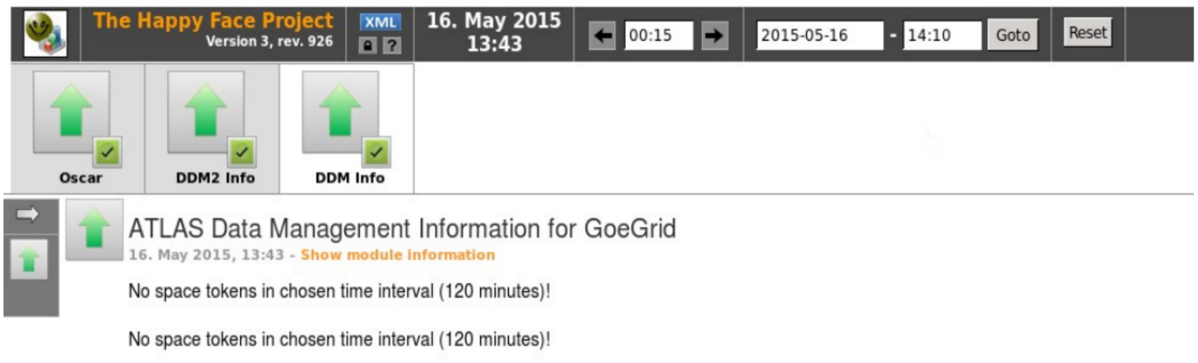


Figure 3.3.: Differences between DQ2 and Rucio.

  - Allowing requests to be queried and cancelled
- Delete replicas from sites
- Ensure dataset consistency on sites
  - In particular manage file losses on sites
- Enforce ATLAS Computing model policies
- Monitor ATLAS data activities

The new system Rucio has many improvements, but it breaks the compatibility with DQ2. One of the central components in DQ2 was the LCG File Catalog (LFC). It is an external replica catalog, responsible for connecting the Physical File Names (PFN) with the Logical File Names (LFN). In Rucio, they can be derived from each other directly via a deterministic function. Therefore, all file replicas produced by ATLAS had to be renamed before the migration to Rucio. This counts up to roughly 300 million files split between about 120 sites with 6 different storage technologies. In order to make this renaming possible, a completely new infrastructure with its own workflow had to be invented [19]. As distinct from DQ2, Rucio now directly supports not only SRM [20], but also GridFTP [21], HTTP and xroot [22]. The main differences between the two systems are graphically displayed in Fig.3.3.



ATLAS Data Management Information for GoeGrid

16. May 2015, 13:43 - Show module information

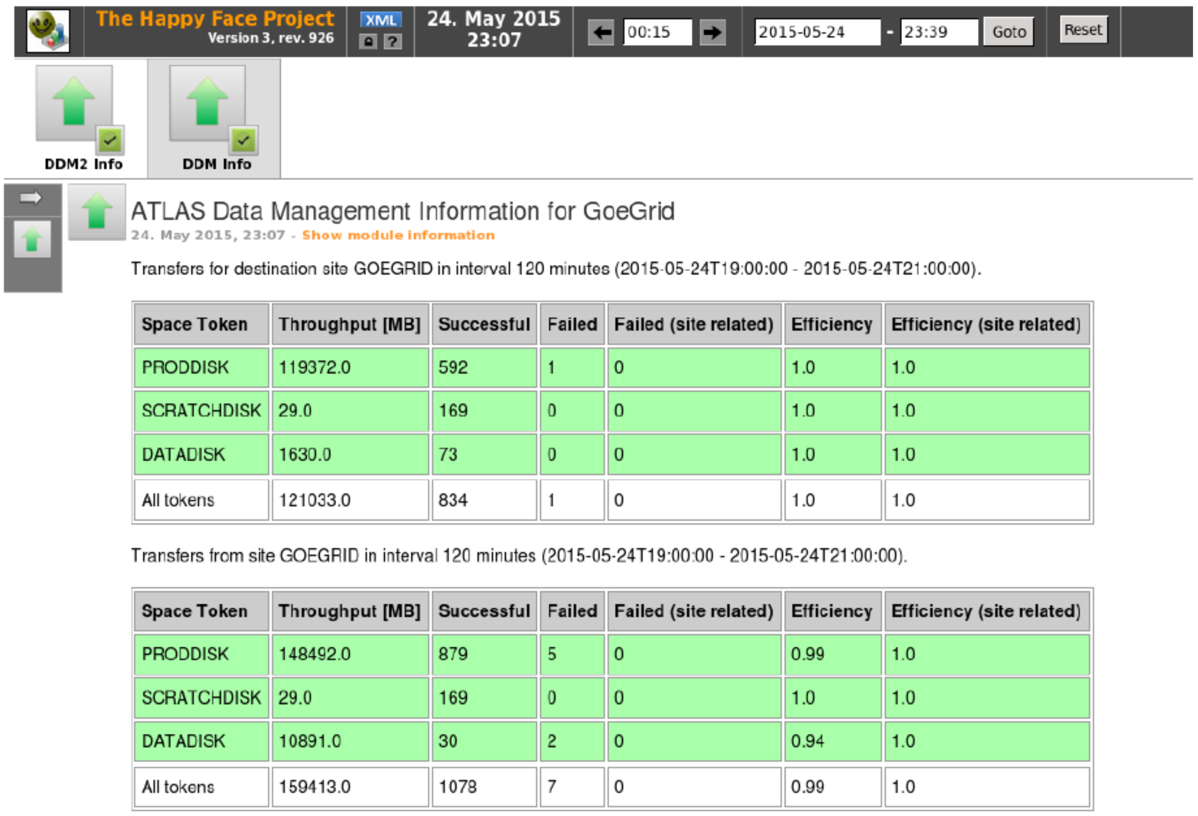
No space tokens in chosen time interval (120 minutes)!

No space tokens in chosen time interval (120 minutes)!

Figure 3.4.: Output of the original DDM module.

3.4.2. DDM module

The DDM module of HappyFace displays information on data transfers from and to the GoeGrid site. Fortunately, the changes caused by the transition to Rucio weakly affected the DDM HappyFace module. The website that displays the monitoring data updated its structure and design. The URL changed from *dashb-atlas-data.cern.ch* to *dashb-atlas-ddm-cern.ch*. This lead to a failure when downloading the JSON files. In Fig.3.4, a screenshot of the output of the original DDM module can be seen.
The first try to fix the problem was enlarging the time interval since the error message stated “No space tokens in chosen time interval (120 minutes)!”. It took a while to identify the actual reason for the failure. Once it was recognised, progress followed quickly. The new URL can simply be specified in the config file of the module. Since not only the URL, but also the parameters for querying the JSON files changed, a few minor changes to the Python script were necessary as well. After these changes, the module was working again. In order to identify possible problems earlier in the future, two new columns were added to the database and filled with the `fromDate` and `toDate` values of the JSON file. This offers additional information for site administrators because the navigation bar in HappyFace only shows the point in time, when `acquire.py` was run for the last time. This date does not specify the actuality of the JSON files themselves (and the information contained in them), but only the actuality of the download. If the monitoring website quit the support and no fresh data were uploaded, this would not be recognised by HappyFace. However, with the new information on the time span of the JSON file, it can be. The output of the new DDM module can be seen in Fig.3.5.



ATLAS Data Management Information for GoeGrid

24. May 2015, 23:07 - Show module information

Transfers for destination site GOEGRID in interval 120 minutes (2015-05-24T19:00:00 - 2015-05-24T21:00:00).

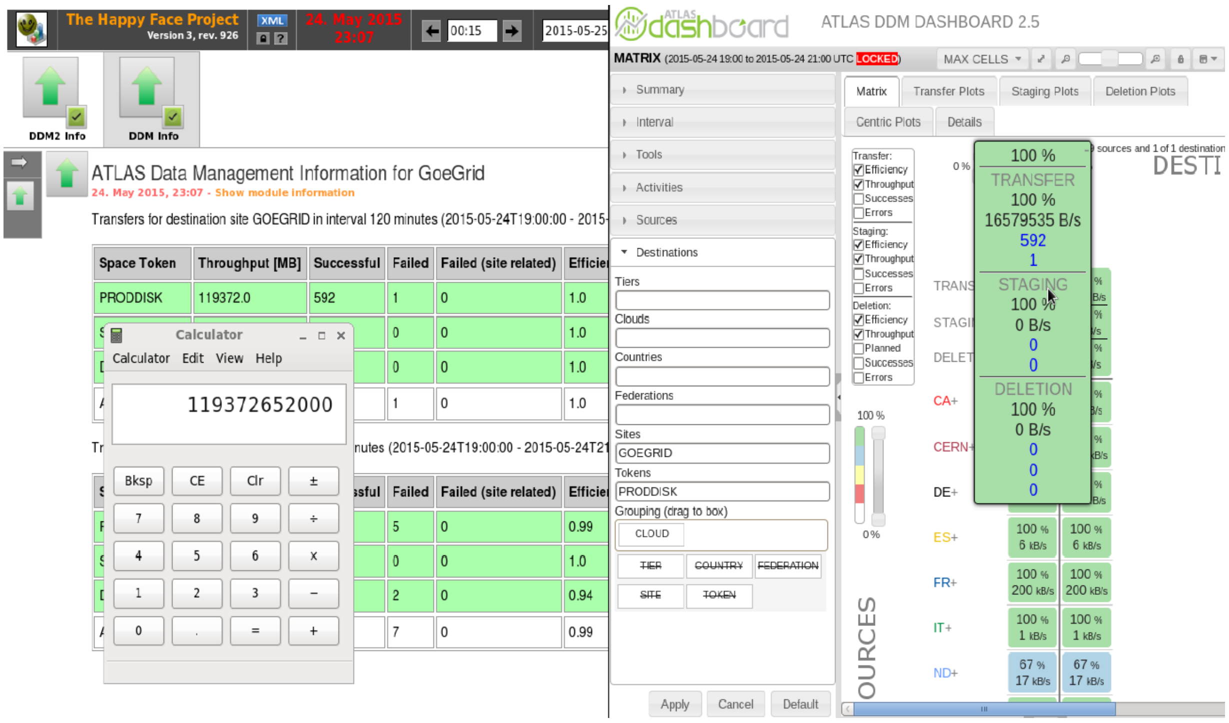
Space Token	Throughput [MB]	Successful	Failed	Failed (site related)	Efficiency	Efficiency (site related)
PRODDISK	119372.0	592	1	0	1.0	1.0
SCRATCHDISK	29.0	169	0	0	1.0	1.0
DATADISK	1630.0	73	0	0	1.0	1.0
All tokens	121033.0	834	1	0	1.0	1.0

Transfers from site GOEGRID in interval 120 minutes (2015-05-24T19:00:00 - 2015-05-24T21:00:00).

Space Token	Throughput [MB]	Successful	Failed	Failed (site related)	Efficiency	Efficiency (site related)
PRODDISK	148492.0	879	5	0	0.99	1.0
SCRATCHDISK	29.0	169	0	0	1.0	1.0
DATADISK	10891.0	30	2	0	0.94	1.0
All tokens	159413.0	1078	7	0	0.99	1.0

Figure 3.5.: Output of the new DDM module.

Since the absolute numbers in the JSON files are not mentioned anywhere else on the DDM website, validating them can be tricky. Doing so is nonetheless necessary because it is not guaranteed that the extracted information really equal the requested. That is the case because the downloaded JSON files are just assumed, but not proven, to be the correct ones. In order to proof this, the data rates on the DDM website can be taken into account. In Fig. 3.6, one can see the comparison between the data rate (DDM website) and the throughput (HappyFace website) of one site. To achieve this, the filters on the DDM website had been set according to the first row of the table displayed by the DDM module. In this case, the destination site was specified as GoeGrid, the token defined as PRODDISK and the time interval set to 120 minutes, reaching from 2015-05-24T19:00:00 to 2015-05-24T21:00:00. The displayed data rate (in units of MB per minute) is the average value for this time interval. Multiplied by 120 minutes, it equals the throughput displayed on the HappyFace website. The result can be seen in the calculator application. This proves the JSON file to be the requested one. From here on, every other entry can be assumed to be correct as well.



The Happy Face Project
Version 3, rev. 926
24. May 2015 23:07
00:15
2015-05-25

DDM2 Info
DDM Info

ATLAS Data Management Information for GoeGrid
24. May 2015, 23:07 - Show module information

Transfers for destination site GOEGRID in interval 120 minutes (2015-05-24T19:00:00 - 2015-

Space Token	Throughput [MB]	Successful	Failed	Failed (site related)	Efficie
PRODDISK	119372.0	592	1	0	1.0
S			0	0	1.0
D			0	0	1.0
A			1	0	1.0

Calculator
Calculator Edit View Help
119372652000

Tr nutes (2015-05-24T19:00:00 - 2015-05-24T21

S	ssful	Failed	Failed (site related)	Efficie
F		5	0	0.99
S		0	0	1.0
D		2	0	0.94
A		7	0	0.99

ATLAS DDM DASHBOARD 2.5
MATRIX (2015-05-24 19:00 to 2015-05-24 21:00 UTC LOCKED)

Destinations
Tiers:
Clouds:
Countries:
Federations:
Sites: GOEGRID
Tokens: PRODDISK
Grouping (drag to box): CLOUD

Transfer: ✓ Efficiency ✓ Throughput Successes Errors
Staging: ✓ Efficiency ✓ Throughput Successes Errors
Deletion: ✓ Efficiency ✓ Throughput Planned Successes Errors

100 %
TRANSFER
100 %
16579535 B/s
592
1
STAGING
100 %
0 B/s
0
0
DELETION
100 %
0 B/s
0
0
0

Figure 3.6.: Verfiying the output of the new DDM module.

3.5. PanDA

3.5.1. Preconditions

The Production and Distributed Analysis (PanDA) system was originally designed for data processing and analysis in the ATLAS experiment. However, scientists all over the world are trying to use it for other data intensive applications as well. It has a highly flexible and scalable architecture, so it can adapt to emerging computing technologies in processing, storing and distributed computing middleware. During the past years, it was very successful in doing so and met all the computing needs of the ATLAS experiment. It is currently managing more than 100 sites, about 5 million jobs per week and roughly 1500 users [23]. It consists of the following parts:

- PanDA server
  - Database back-end
  - Brokerage
  - Dispatcher
- PanDA pilot system
  - Job wrapper
  - Pilot factory

- Dynamic data placement
- Information system
- Monitoring system

The increasing interest in the PanDA system by other big data sciences lead to a proposal of the "Next Generation Workload Management and Analysis System for Big Data". The new project, called BigPanDA, has been started in 2012 and the development process is still ongoing. Although ATLAS remains the most important user community, BigPanDA will encompass a set of software packages for many different experiments. There will be a general core and usage specific code that is encapsulated into plug-ins [23]. This modular design allows easier adaptation to the respective needs.
The website displaying the monitoring data changed its structure and design completely. It now allows a clear separation between data access and visualisation. Another new feature is the modular design which encourages to bring up new data transfer objects. The URL changed from *panda.cern.ch* to *bigpanda.cern.ch*.

3.5.2. PanDA module

The PanDA module displays site and queue specific information on jobs. These jobs can either be production or analysis type. Originally, the amount of active, running, defined, holding, finished, failed and cancelled jobs for each queue were shown.
Since the website providing the sources changed drastically because of the transition to BigPanda, massive changes to the PanDA module were necessary. In the original module, only the names and statuses of the queues were extracted from JSON files. These JSON files changed their content and their location. They are now downloaded from the ATLAS Grid Information System (AGIS). The algorithm to extract the data had to be updated. The amounts of jobs for each of the properties mentioned above were extracted from tables in HTML files. In the new module, these numbers are retrieved via JSON files from the BigPanDA website. They can, however, not be accessed via the URL directly, but only with the help of the Linux command `wget` in combination with a header that specifies the acceptance JSON files. In order to implement this, the download service had to be modified. The data extraction had to be developed from scratch since it simply did not exist before. In addition to this, worker node information graphs are now displayed in the module as well.

New download method

As a first try to solve the problem, JSON files were manually downloaded and saved via the `wget` command. They were then read in by the actual module. Since this completely contradicted the concept of HappyFace, it became clear that the download had to be somehow included into the workflow of HappyFace.
Because the HappyCore should remain untouched, a new module was created with the sole purpose of downloading the JSON files from BigPanDA. It inherited from `hf.download Service` and had no corresponding HTML file as it did not generate any output. The downloading function within the new module was then called inside the PanDA module. However, this did not work out. The reason for this lays in the rather complex structure of `acquire.py`. In this script, not only functions of the modules are executed, but also functions of the HappyCore. In order to download the JSON files successfully, the download command has to be executed by `acquire.py`. To do that, the modules folder was imported at the beginning of `acquire.py`. Importing a folder requires a file called `__init__.py` in it. This file makes Python treat the directory as a containing package. After creating it and executing the new download method by `acquire.py`, new errors occurred concerning archives in the database. Both download methods (the original and the new one) tried to create a new archive corresponding to the point in time when `acquire.py` was run. The problem might have been solved by changing the name of one of those archives, but that would also have made changes to the data extraction from the database necessary. Because of these difficulties (and because the HappyCore did not remain untouched anyway), the decision was made, to discard the idea of a download module. Instead, `hf.downloadService` should be modified directly.
An extract from the modified download service, `downloadservice.py`, can be seen in Listing 3.7. The recently added part (*ll.* 4-6) has the same structure as the already existing one containing the `wget` command without any header (*ll.* 7-9). At the beginning of the new code, it is examined, whether the additional header is necessary (*l.* 4). That is the case if the URL starts with bigpanda and is either pointing to a job or a task list. If so, the actual download command uses the additional header (*l.* 5). The directory and name of the output document are the same in all cases. For a better overview, 14 lines of code have been omitted between the blocks of code (*ll.* 6, 9). They are responsible for printing error messages in case the download failed. With the new download service, JSON files can be retrieved from BigPanDA.

```
if self.file.url.startswith("file://"):
    path = self.file.url[len("file://"):]
    shutil.copy(path, self.file.getTmpPath(True))
elif self.file.url.startswith("http://bigpanda.cern.ch/jobs") or self.file.url.startswith("http://bigpanda.cern.ch/tasks"):
    command = "wget␣--header='Accept:␣application/json'␣--header='Content-Type:␣application/json'␣--output-document=\"%s\"␣%s␣%s␣\"%s\"" % (self.file.getTmpPath(True), "" if self.file.config_source == "local" else self.global_options, self.file.options, self.file.url)
    #[...] omitted code
else:
    command = "wget␣--output-document=\"%s\"␣%s␣%s␣\"%s\"" % (self.file.getTmpPath(True), "" if self.file.config_source == "local" else self.global_options, self.file.options, self.file.url)
    #[...] omitted code
```

Listing 3.7: `Code extract from downloadservice.py`.

Data extraction

Extracting the data takes place in separate classes within the python script of the module. Usually, there is one class for each type of source file. In the new PanDA module, there are `cloud_class2` for extracting the queue names and queue statuses and `bigpanda_class` for extracting information from the JSON files from BigPanDA. Every class contains multiple functions. The functions normally return one specific value each. Listing 3.8 shows the extract from the function `get_queue_status` that is responsible for the production queues. It returns a list of production queue names and their statuses. To do so, a `for` loop runs through every queue listed in the JSON file (*l.* 1). In case that queue matches the parameter that this function receives (*l.* 2), another `for` loop runs through all the sub queues (*l.* 3). Theses sub queues are then appended to the list `production_queues` (*l.* 4). The status of the corresponding queue is also queried and returned afterwards (*ll.* 5-11). The whole function `get_queue_status` can be seen in the appendix (Listing A.1).

```
 1  for item in range(len(self.json_source)):
 2      if queue == self.json_source[item]["panda_resource"]:
 3          for i in range(len(self.json_source[item]["queues"])):
 4              production_queues.append("%s␣%s"%(str(self.json_source[item]["queues"][i]["
                    ce_endpoint"]),str(self.json_source[item]["status"])))
 5              if self.json_source[item]["status"]=="online":
 6                  queue_status="online"
 7              elif queue_status!="online" and self.json_source[item]["status"]!="online":
 8                  queue_status=self.json_source[item]["status"]
 9       else:
10           continue
11  return production_queues,queue_status
```

Listing 3.8: Code extract from `get_queue_status`.

Another example of a data extraction function is `get_numberof_activated_jobs`. It extracts the number of activated jobs from a job list that was retrieved from the BigPanDA website. Its simple code is displayed in Listing 3.9. After resetting the amount of activated jobs (*l.* 1), a `for` loop iterates over all listed jobs (*l.* 3). If the job status is "activated" (*l.* 4), the amount of activated jobs is increased by one (*l.* 5). At the end, the result is returned (*l.* 6). The same structure is used for the other types of jobs.

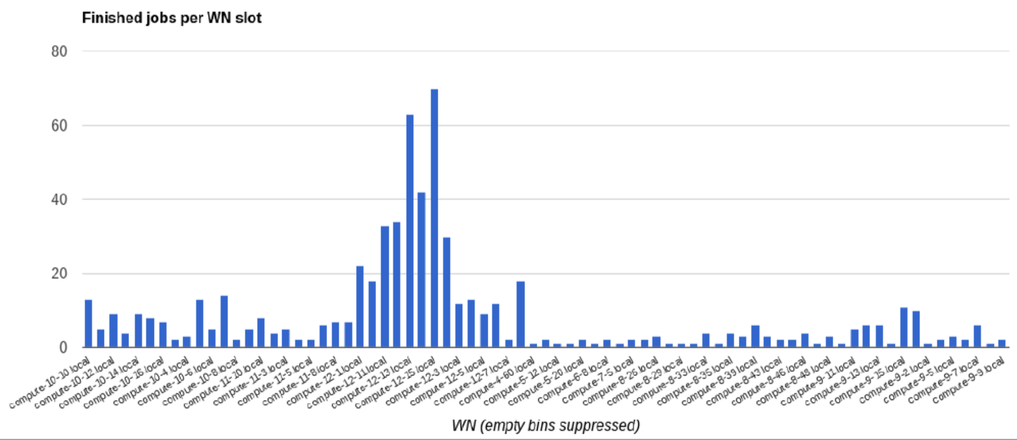
```
1  def get_numberof_activated_jobs(self):
2      activated_jobs=0
3      for i in range(len(self.json_source)):
4          if self.json_source[i]["jobstatus"]=="activated":
5              activated_jobs+=1
6      return activated_jobs
```

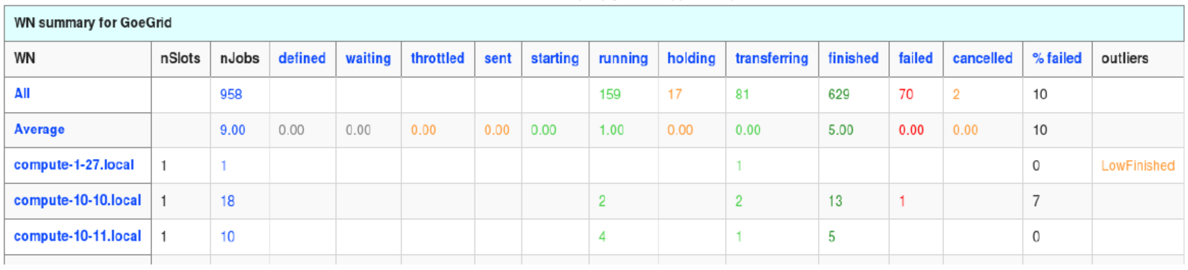
Listing 3.9: Code of `get_numberof_activated_jobs`.

In order to be able to observe an arbitrary amount of sites and queues, all data extraction functions are executed within `for` loops. Some corresponding code is shown in Listing 3.10. `site_names` and `queue_names` are lists that contain all sites and queues defined in the config file, respectively. In the `for` loop iterating over all sites (*l.* 1), `cloud_class2` is executed with the current time and the JSON file from AGIS as parameters (*l.* 2). A second `for` loop iterates over all source files from BigPanDA and queues (*l.* 3). (`wns` has something to do with worker nodes and will be explained later.) After opening the downloaded files (*ll.* 5-6), the data extraction functions are executed. The same `for` loops are used to create subtables and to access the database. This concept already existed in the old PanDA module and also comes into play in other modules such as DDM.

```
1   for site in self.site_names:
2       grid_site_info = cloud_class2(datetime.now(),schedconfig_content)
3           for source, queue, wns in map(None, (self.site_sources[site])['analysis'], (self.
                queue_names[site])['analysis'], self.wns_sources[site]['analysis']):
4               queue_info = {}
5               source_content = open(source.getTmpPath()).read()
6               wns_content = open(wns.getTmpPath()).read()
7               queue_info['wns_failed'] = grid_site_info.get_wns_failed(wns_content)
8               queue_info['wns_finished'] = grid_site_info.get_wns_finished(wns_content)
9               bigpanda_info = bigpanda_class(datetime.now(),source_content)
10              analysis_info=grid_site_info.get_queue_status(queue)
11              queue_info['site_name'] = site
12              queue_info['queue_name'] = queue
13              queue_info['queue_link'] = self.panda_analysis_url.split('|')[2]+queue+'&
                    hours='+self.panda_analysis_interval
14              queue_info['queue_type'] = 'analysis'
15              queue_info['status'] = analysis_info[1]
16              queue_info['active_jobs'] = bigpanda_info.get_numberof_activated_jobs()
17              queue_info['running_jobs'] = bigpanda_info.get_numberof_running_jobs()
```

Listing 3.10: Code extract form `Panda.py`: Data extraction in `for` loops.





WN summary for GoeGrid

WN	nSlots	nJobs	defined	waiting	throttled	sent	starting	running	holding	transferring	finished	failed	cancelled	% failed	outliers
All		958						159	17	81	629	70	2	10	
Average		9.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	5.00	0.00	0.00	10	
compute-1-27.local	1	1								1				0	LowFinished
compute-10-10.local	1	18						2		2	13	1		7	
compute-10-11.local	1	10						4		1	5			0	

Figure 3.7.: Worker node graph and table on the BigPanDA website.

Worker node information graphs

In contrast to its predecessor, the BigPanDA website offers worker node information. This indicates, how many jobs finished (or failed) for each worker node. The same information is displayed in a table and a histogram. In Fig.3.7, a corresponding screenshot is displayed. Since the worker node graphs provide a good overview of the condition of each queue, the decision was made to display them via the PanDA module. Unfortunately, there are no JSON files available for that purpose. The only possibility to retrieve the needed data is downloading the HTML files, which will be referred to as `wn.html` in the following. BigPanDA uses the *google visualization* API to create the plots. The code extract from `wn.html` which is responsible for plotting is shown in Listing 3.11. The data to plot is a string within the HTML file. The single data points are embedded in square brackets and consist of the name of the worker node and a corresponding number.

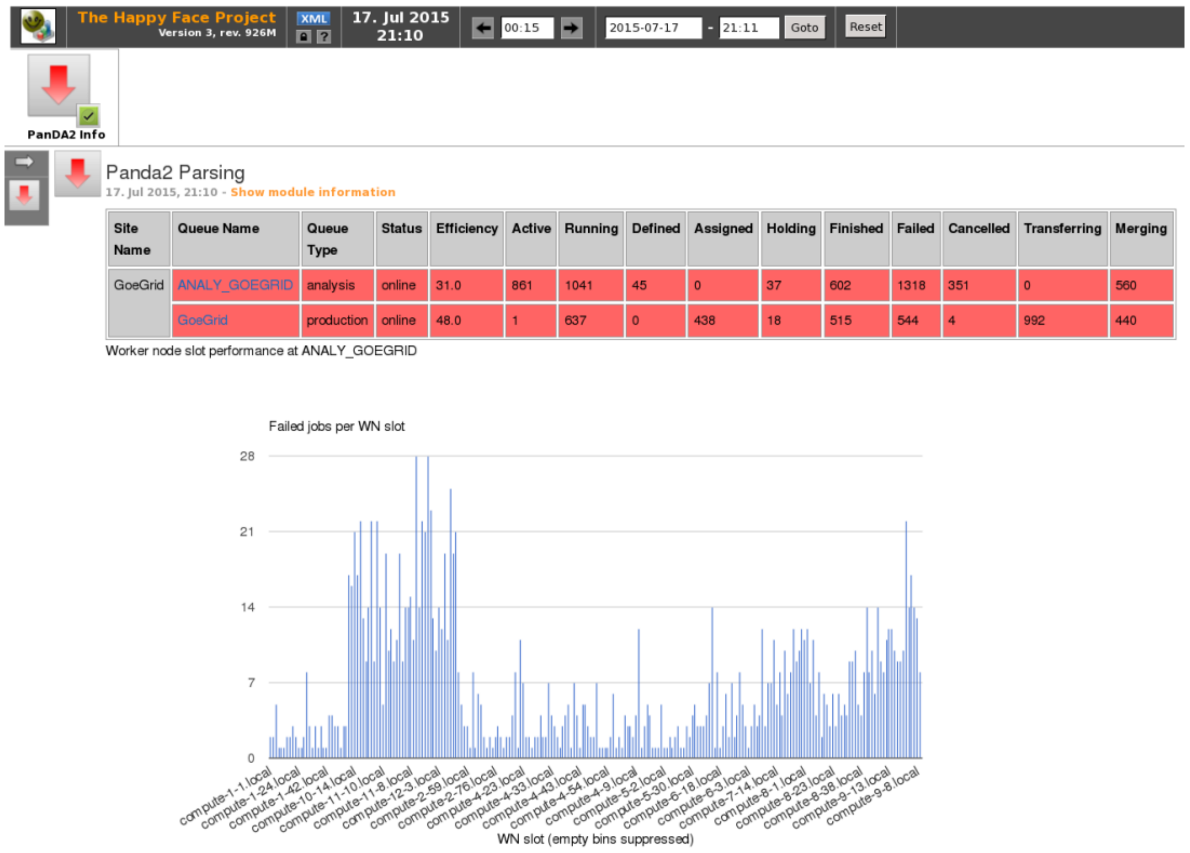
```
<script type="text/javascript" src="https://www.google.com/jsapi"></script>
<script type="text/javascript">
  google.load("visualisation", "1", {packages:["corechart"]});
  google.setOnLoadCallback(drawChart);
  function drawChart() {

    var data = google.visualization.arrayToDataTable([
      ['Time', 'Count'],


      ['compute-10-10.local', 1],

      ['compute-10-12.local', 1],

      ['compute-10-16.local', 1],

      ['compute-11-8.local', 2],
```

Listing 3.11: Code extract from `wn.html`.

In order to create the same plots via the PanDA module, the string which contains the data points has to be extracted from `wn.html`. To achieve that, the file has to be cut at `var data = google.visualization.arrayToDataTable([` and a similar expression which marks the end of the plotted data. This string is then inserted into the same google visualization command again - this time inside the HTML file of the PanDA module. Of course, all of this is taking place within proper `for` loops. It is also important to iterate the names of the generated plot. Otherwise the API tries to create several plots with the same name, which inevitably leads to an error. The code extract from `Panda.html` that is responsible for plotting via the google visualization is displayed in the appendix (Listing A.3). The output of the PanDA module using google visualization is displayed in Fig.3.8. In order to save space, there is only one plot shown, but the other ones were right below it.

Plotting with the help of the google visualization can be done easily and generates nice results. However, it violates the rules of data privacy which is why it cannot be used in the final version of the module. Nonetheless, this method was shown briefly since it is also used by the offical PanDA website.

As an alternative and proper method, `pyplot` from the `maplotlib` can be used. Since the data to plot cannot be read in as a string in this case, the data extraction became more complicated. The responsible part of source code for that can be seen in Listing 3.12. The consigned data (*l.* 1), `wns_content`, is the string which was cut out of `wn.html`. The other parameters of the function are necessary to specify the destination of the output file. The first line of the string contains the names of the columns and has to be cut off (*l.* 5). Then, all space bars and tabulators are deleted (*ll.* 6-8). After transforming the



Panda2 Parsing

17. Jul 2015, 21:10 - Show module information

Site Name	Queue Name	Queue Type	Status	Efficiency	Active	Running	Defined	Assigned	Holding	Finished	Failed	Cancelled	Transferring	Merging
GoeGrid	ANALY_GOEGRID	analysis	online	31.0	861	1041	45	0	37	602	1318	351	0	560
	GoeGrid	production	online	48.0	1	637	0	438	18	515	544	4	992	440

Worker node slot performance at ANALY_GOEGRID

Figure 3.8.: Displaying worker node graphs via the google visualization API.

string into an array by splitting it up at every comma (*l.* 9), a `for` loop iterates over all its entries (*l.* 10). Within this loop, every second entry is appended to the list of names, `ind` (*ll.* 11-12). The other entries are appended to the list `numbers` (*ll.* 13-14). By that time, `ind` contains the names of the worker nodes and `numbers` the corresponding numbers of jobs (finished or failed). Both lists, however, still have unwanted characters in it as well. These are deleted (*ll.* 15-16), and the entries of `numbers` are converted from strings to integers (*ll.* 18-19). The omitted code (*ll.* 2, 20) contains plotting commands from the matplotlib, which are not of interest here. The variable `dest` describes the destination of the resulting output file. This is crucial, since it has to be specified in the HTML file of the module in order to display the plot. The complete code can be seen in the appendix (Listing A.2).

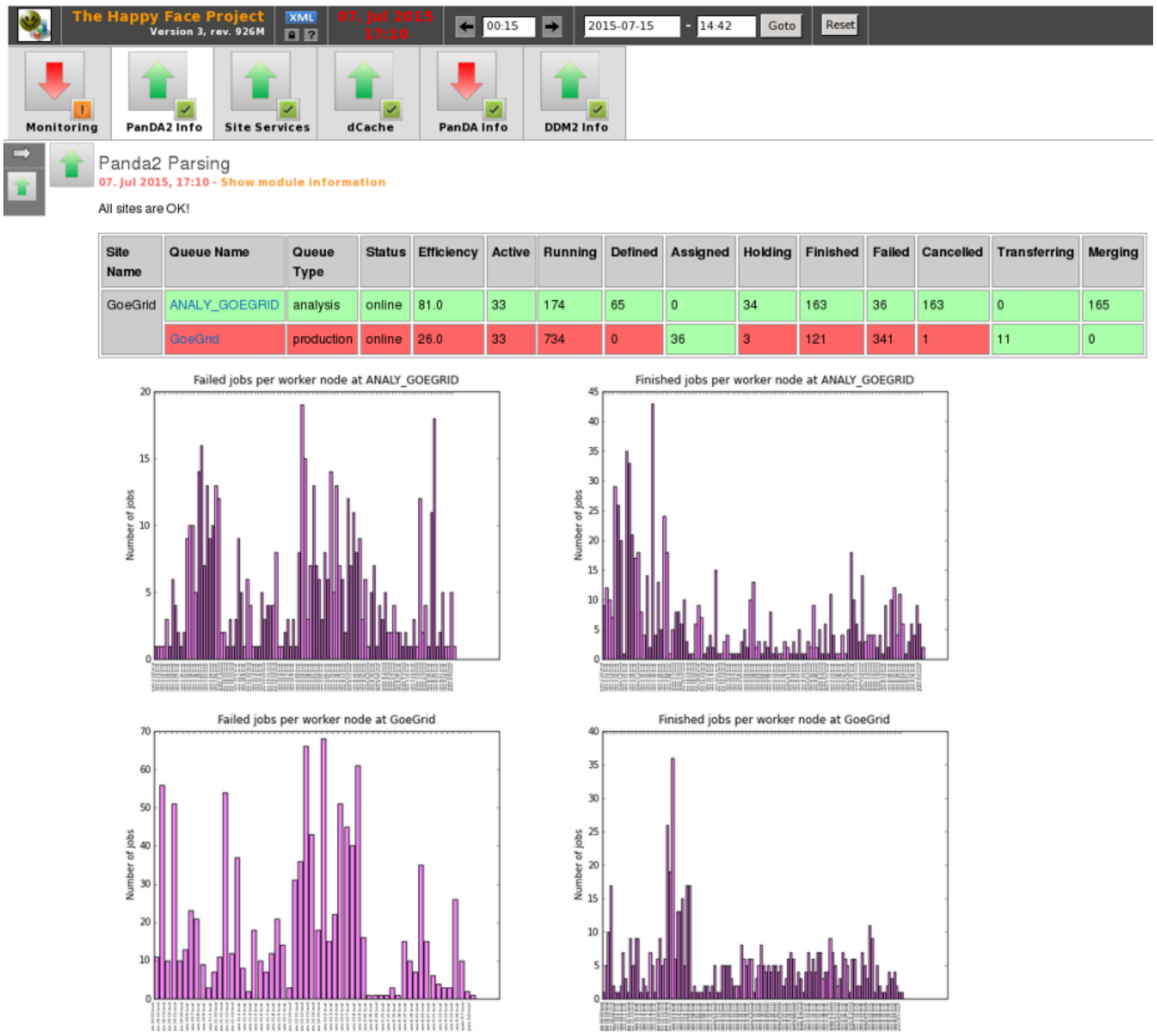
```
def wns_plot(self,queue,wns_content,option,a,b):#option='Failed' or 'Finished', a and b to get self.run and self.instance_name
    #[...] omitted code
    ind = ""
    numbers = ""
    ind_failed = wns_content.split("'Count'],")[1]# Cut off names of variables
    ind_failed = ind_failed.replace("\n", "")# Delete all new lines, tabulators and spacebars
    ind_failed = ind_failed.replace("\t", "")
    ind_failed = ind_failed.replace("␣", "")
    ind_failed = ind_failed.split(',')# Create array out of string
    for i in range(len(ind_failed)):
        if i%2==0:# Iterate over all even entries (name of wn)
            ind = ind + ind_failed[i]
        else:# Iterate over all odd entries (number of jobs)
            numbers = numbers + ind_failed[i]
    ind = ind.split("'['")
    numbers = numbers.split("]")
    numbers = numbers[0:(len(numbers)-1)]# Cut off last entry as it's always empty
    for i in range(len(numbers)):
        numbers[i]=int(numbers[i])
    #[...] omitted code
    dest = "/"+hf.downloadService.getArchivePath(a, b+"_"+queue+"_"+option+'.png')
    return dest
```

Listing 3.12: Code extract from `Panda.py`: Extracting worker node information.

```
1      % for site in module.dataset['site_names'].split(','):
2          % for detail in site_details:
3              % if detail['site_name'] == site:
4  <img src="${detail['filename_failed_plot']}" alt=""/>
5  <img src="${detail['filename_finished_plot']}" alt=""/>
6              % endif
7          % endfor
8      % endfor
```

Listing 3.13: Code extract from `Panda.html`: Displaying plots.

In Listing 3.13, the code extract that is responsible for displaying the previously generated plots can be seen. Within proper loops, the destination of the plots is retrieved from the database (*ll.* 4-5). The final outcome of the new PanDA module is displayed as a screenshot in Fig 3.9.


The Happy Face Project
Version 3, rev. 926M
XML
07. Jul 2015 17:10
00:15
2015-07-15
14:42
Goto
Reset
Monitoring
PanDA2 Info
Site Services
dCache
PanDA Info
DDM2 Info
Failed jobs per worker node at ANALY_GOEGRID
Finished jobs per worker node at ANALY_GOEGRID
Failed jobs per worker node at GoeGrid
Finished jobs per worker node at GoeGrid
Number of jobs

Panda2 Parsing

07. Jul 2015, 17:10 - Show module information

All sites are OK!

Site Name	Queue Name	Queue Type	Status	Efficiency	Active	Running	Defined	Assigned	Holding	Finished	Failed	Cancelled	Transferring	Merging
GoeGrid	ANALY_GOEGRID	analysis	online	81.0	33	174	65	0	34	163	36	163	0	165
	GoeGrid	production	online	26.0	33	734	0	36	3	121	341	1	11	0

Figure 3.9.: Output of the final PanDA module.

4. Conclusion and Outlook

4.1. Conclusion

Before the modification, both modules could not be used by scientists. The data extraction of the DDM module failed and the only output on the HappyFace website was an error message. The PanDA module generated output, but the data were retrieved from an outdated source and therefore not valid anymore. Since the modification, both modules have been working properly. The new DDM module also displays the time span that is covered by the JSON files. The new PanDA module additionally displays worker node information graphs and the amount of three new job states for each queue and site.
In the end, the update of the modules was successful. The speed of progress was constantly increasing because getting to know the HappyFace programme took much time. After identifying the problems and understanding the principles of HappyFace, solutions were implemented quickly. With more time to work, modules could have been updated more efficiently.

4.2. Outlook

Beside the two modules that were modified during the working period of this thesis, there are other modules that need to be updated as well. Rapid developments in computing technologies lead to a constantly evolving soft- and hardware setup of the WLCG. In consequence of this, HappyFace modules will always have to be adjusted.
There are, however, new developments that are not concerned with module development, like, for example, the HappyFace smartphone application that was developed by Fabian Kukuck [24].
All in all, the HappyFace project is constantly evolving and will prospectively enable many scientists to have a compact view on the grid.

A. Appendix

```python
def get_queue_status(self,queue):
    queue_status="unknown"
    analysis_queues=[]
    production_queues=[]
    if "analy" in queue.lower():
        for item in range(len(self.json_source)):
            if queue == self.json_source[item]["panda_resource"]:
                for i in range(len(self.json_source[item]["queues"])):
                    if str(self.json_source[item]["queues"][i]["ce_endpoint"])!="Unknown" and str(self.json_source[item]["queues"][i]["ce_endpoint"])!="to.be.set":
                        analysis_queues.append("%s %s"%(str(self.json_source[item]["queues"][i]["ce_endpoint"]),str(self.json_source[item]["status"])))
                    else:
                        analysis_queues.append("%s %s"%(str(self.json_source[item]["panda_resource"]),str(self.json_source[item]["status"])))
                    if self.json_source[item]["status"]=="online":
                        queue_status="online"
                    elif queue_status!="online" and self.json_source[item]["status"]!="online":
                        queue_status=self.json_source[item]["status"]
            else:
                continue
        return analysis_queues,queue_status
    else:
        for item in range(len(self.json_source)):
            if queue == self.json_source[item]["panda_resource"]:
                for i in range(len(self.json_source[item]["queues"])):
                    production_queues.append("%s %s"%(str(self.json_source[item]["queues"][i]["ce_endpoint"]),str(self.json_source[item]["status"])))
                    if self.json_source[item]["status"]=="online":
                        queue_status="online"
                    elif queue_status!="online" and self.json_source[item]["status"]!="online":
                        queue_status=self.json_source[item]["status"]
            else:
                continue
        return production_queues,queue_status
    return queue_status
```

Listing A.1: Code extract from Panda.py: `get_queue_status`.

```
def wns_plot(self,queue,wns_content,option,a,b):#option='Failed' or 'Finished', a and b
    to get self.run and self.instance_name
    import matplotlib
    import matplotlib.pyplot as plt
    self.plt = plt
    ind = ""
    numbers = ""
    ind_failed = wns_content.split("'Count'],")[1]# Cut off names of variables
    ind_failed = ind_failed.replace("\n", "")# Delete all new lines, tabulators and
        spacebars
    ind_failed = ind_failed.replace("\t", "")
    ind_failed = ind_failed.replace("␣", "")
    ind_failed = ind_failed.split(',')# Create array out of string
    for i in range(len(ind_failed)):
        if i%2==0:# Iterate over all even entries (name of wn)
            ind = ind + ind_failed[i]
        else:# Iterate over all odd entries (number of jobs)
            numbers = numbers + ind_failed[i]
    ind = ind.split("'['")
    numbers = numbers.split("]")
    numbers = numbers[0:(len(numbers)-1)]# Cut off last entry as it's always empty
    for i in range(len(numbers)):
        numbers[i]=int(numbers[i])

    fig_abs = self.plt.figure()
    plt.bar(range(len(numbers)), numbers, color='violet')
    plt.xticks(range(len(numbers)), ind, size=5, rotation=90)
    plt.show()
    axis_abs = fig_abs.add_subplot(111)
    axis_abs.set_ylabel('Number␣of␣jobs')
    axis_abs.set_xlabel('Name␣of␣WN')
    axis_abs.set_title(option+'␣jobs␣per␣worker␣node␣at␣'+queue)
    fig_abs.savefig(hf.downloadService.getArchivePath(a, b+"_"+queue+"_"+option+'.png'),
        dpi=60)
    plt.close()
    dest = "/"+hf.downloadService.getArchivePath(a, b+"_"+queue+"_"+option+'.png')
    return dest
```

Listing A.2: Code extract from `Panda.py`: Extracting worker node information.

```
<!--Alternative plotting via Google Visualisation-->

    % for site in module.dataset['site_names'].split(','):
        % for detail in site_details:
            % if detail['site_name'] == site:

                <div class="large-12␣columns">
<div class='section'>Worker node slot performance  at ${detail['queue_name']}</div>
    <script type="text/javascript" src="https://www.google.com/jsapi"></script>
    <script type="text/javascript">
      google.load("visualization", "1", {packages:["corechart"]});
      google.setOnLoadCallback(drawChart);
      function drawChart() {
        var data${detail['queue_name']} = google.visualization.arrayToDataTable([${detail
            ['wns_failed']}])
        var options = {
          title: 'Failed jobs per WN slot',
          legend: { position: 'none' },
          hAxis: {title: 'WN slot (empty bins suppressed)', titleTextStyle: {color: '
              black'},  textStyle: { fontSize:11},},
        };

        var chart${detail['queue_name']} = new google.visualization.ColumnChart(document.
            getElementById("${detail['queue_name']}1"));
        chart${detail['queue_name']}.draw(data${detail['queue_name']}, options);

        var data2${detail['queue_name']} = google.visualization.arrayToDataTable([${
            detail['wns_finished']}])
        var options2 = {
          title: 'Finished jobs per WN slot',
          legend: { position: 'none' },
          hAxis: {title: 'WN (empty bins suppressed)', titleTextStyle: {color: 'black'},
               textStyle: { fontSize:11},},
        };

        var chart2${detail['queue_name']} = new google.visualization.ColumnChart(document
            .getElementById("${detail['queue_name']}2"));
        chart2${detail['queue_name']}.draw(data2${detail['queue_name']}, options2);

      }
    </script>

    <div id="${detail['queue_name']}1" style="height:␣500px;"></div>
    <div id="${detail['queue_name']}2" style="height:␣500px;"></div>

            % endif
        % endfor
    % endfor
% endif
</%def>
```

Listing A.3: HTML code of alternative plotting via goole visualization.

Bibliography

[1] L. Evans, P. Bryant, *LHC Machine*, JINST **3**, S08001 (2008)

[2] J. Andreeva, M. Boehm, S. Belov, J. Casey, F. Dvorak, et al., *Job monitoring on the WLCG scope: Current status and new strategy*, J.Phys.Conf.Ser. **219**, 062002 (2010)

[3] A. Collaboration (ATLAS), *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST **3**, S08003 (2008)

[4] B. Lemmer, *Measurement of Spin Correlations in $t\bar{t}$ Events from pp Collisions at $\sqrt{s}$ = 7 TeV in the Lepton + Jets Final State with the ATLAS Detector (Dissertation at Georg-August-University Göttingen, II.PHYSIK-UNIGÖ-DISS-2014-02)* (2014)

[5] M. Keil, *Pixeldetektoren aus Silizium und CVD-Diamant zum Teilchennachweis in* ATLAS *bei* LHC *(Dissertation at Rheinische Friedrich-Wilhelms-University Bonn), URN: urn:nbn:de:hbz:5n-00564* (2001)

[6] I. Bird, *Computing for the Large Hadron Collider*, Annual Review of Nuclear and Particle Science **61(1)**, 99 (2011)

[7] CERN, *The Grid: A system of tiers* (2015), URL `home.web.cern.ch/about/computing/grid-system-tiers`

[8] J. Meyer, A. Quadt, P. Weber (ATLAS), *ATLAS Tier-2 at the compute resource center GoeGrid in Goettingen*, J.Phys.Conf.Ser. **331**, 072055 (2011)

[9] C. Ay, J. Meyer, A. Quadt, C. Boehme, O. Haan, U. Schwardmann, *Das Goettinger Grid-Ressourcen-Zentrum GoeGrid, Grid-Technologie in Goettingen*, GWDG-Bericht **74** (2009)

[10] M. Massie, B. Chun, D. Culler, *The ganglia distributed monitoring system: design, implementation, and experience*, Parallel Computing **30** (2004)

[11] A. B. M. Moniruzzaman, S. A. Hossain, *Comparative Study on Agile software development methodologies*, CoRR **abs/1307.3356** (2013), URL `http://arxiv.org/abs/1307.3356`

[12] A. Carpen-Amarie, M. I. Andreica, V. Cristea, *An Algorithm for File Transfer Scheduling in Grid Environments*, CoRR **abs/0901.0291** (2009), URL `http://arxiv.org/abs/0901.0291`

[13] J. Andreeva, A. Beche, S. Belov, I. Kadochnikov, P. Saiz, et al., *WLCG Transfers Dashboard: a Unified Monitoring Tool for Heterogeneous Data Transfers*, J.Phys.Conf.Ser. **513**, 032005 (2014)

[14] V. Mauch, et al., *The HappyFace Project*, in *Journal of Physics: Conference Series*, volume 331, page 082011, IOP Publishing (2011)

[15] V. Buge, V. Mauch, G. Quast, A. Scheurer, A. Trunov, *Site specific monitoring of multiple information systems: The HappyFace project*, J.Phys.Conf.Ser. **219**, 062057 (2010)

[16] C. G. Wehrberger, *HappyFace Meta-Monitoring für* Atlas *im Worldwide* Lhc *Computing Gird (Master thesis at Georg-August-University Göttingen, II.Physik-UniGö-MSc-2013/07)* (2013)

[17] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta, *Comparison of JSON and XML Data Interchange Formats: A Case Study*, pages 157–162, CAINE (2009), URL `http://www.mendeley.com/research/comparison-json-xml-data-interchange-formats-case-study-4/`

[18] V. Garonne, G. A. Stewart, M. Lassnig, A. Molfetas, M. Barisits, et al., *The ATLAS Distributed Data Management project: Past and future*, J.Phys.Conf.Ser. **396**, 032045 (2012)

[19] C. Serfon, et al. (ATLAS), *ATLAS DQ2 to Rucio renaming infrastructure*, J.Phys.Conf.Ser. **513**, 042008 (2014)

[20] A. Shoshani, A. Sim, J. Gu, *Storage Resource Managers*, in J. Nabrzyski, J. Schopf, J. Weglarz, editors, *Grid Resource Management*, volume 64 of *International Series in Operations Research Management Science*, pages 321–340, Springer US (2004)

[21] W. E. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. T. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, *Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing*, CoRR **cs.DC/0103022** (2001), URL `http://arxiv.org/abs/cs.DC/0103022`

[22] S. de Witt, A. Lahiff, *Quantifying XRootD Scalability and Overheads*, Journal of Physics: Conference Series **513(3)**, 032025 (2014)

[23] T. Maeno, et al. (ATLAS), *Evolution of the ATLAS PanDA workload management system for exascale computational science*, J. Phys. Conf. Ser. **513**, 032062 (2014)

[24] F. Kukuck, *Erstellen und Veröffentlichen einer neuen Smartphone Application für das HappyFace Meta-Monitoring Tool (Bachelor thesis at Georg-August-University Göttingen)* (2014)

Acknowledgement

Special thanks to Prof. Dr. Arnulf Quadt who gave me the opportunity to write this thesis. I am also deeply thankful to Haykuhi Musheghyan and Dr. Gen Kawamura who always welcomed me at their office and answered all of my questions.

Erklärung nach §13(9) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig 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.

Göttingen, den 2. Juni 2016

(Lino Oscar Gerlach)