GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Fakultät für
Physik

Bachelor's Thesis

Modulentwicklung im Rahmen des HappyFace-Projektes für ATLAS Grid Computing

Module developments on the HappyFace project for the ATLAS Grid Computing

prepared by

Eric Buschmann

from Krefeld

at the II. Physikalischen Institut

Thesis number: II.Physik-UniGö-BSc-2014/02

Thesis period: 25th October 2013 until 11th February 2014

First referee: Prof. Dr. Arnulf Quadt

Second referee: Priv.Doz. Dr. Jörn Große-Knetter

Zusammenfassung

Um die beim ATLAS-Experiment am LHC anfallenden Daten zu verarbeiten, ist eine große Grid-Infrastruktur notwendig, die weltweit verteilt betrieben wird. Um einen ausfallfreien Betrieb und hohe Produktivität zu gewährleisten, sind leistungsfähige Überwachungssysteme notwendig, die es erlauben, Fehler schnell zu identifizieren und zu beseitigen. Am ATLAS Tier-2-Zentrum GoeGrid kommt hierfür unter anderem das Meta-Monitoring-Tool HappyFace zum Einsatz. Im Rahmen dieser Bachelorarbeit wurden hierfür zwei Module entwickelt, die es ermöglichen, die im Cluster laufenden Prozesse zu überwachen und dabei auftretende Fehler zu identifizieren. Diese Module sammeln Daten von den PBS und CREAM CE Diensten und bereiten diese auf, um detaillierte und nach verschiedenen Kriterien aufgeschlüsselte Statusinformationen möglichst in Echtzeit bereitzustellen. Inzwischen sind diese Module öffentlich verfügbar und können so auch von anderen Rechenzentren genutzt werden.

Abstract

A globally distributed computing infrastructure is employed to process the data generated by the ATLAS experiment at LHC. To guarantee stable running Grid services and high efficiency by quickly identifying and rectifying failures, capable monitoring systems are necessary. The ATLAS Tier 2 centre GoeGrid utilises, among others, the meta-monitoring tool HappyFace. In the course of this Bachelor's Thesis, two modules for monitoring jobs running in the cluster and identifying errors were developed. These modules collect and process data from the PBS and CREAM CE services in order to provide detailed status information itemised by several distinct criteria. By now, both modules are publicly available and may therefore be used by other sites.

Keywords: WLCG, ATLAS, Grid Computing, GoeGrid, HappyFace, Meta-Monitoring, CREAM CE, PBS

Contents

1. Introduction

The Large Hadron Collider (LHC) [1], a proton-proton collider with a design centre-of-mass energy of 14 TeV, at CERN [2] hosts the four main experiments ALICE [3], ATLAS [4], CMS [5] and LHCb [6].

The discovery of a Higgs-like boson at ATLAS [7] and CMS [8] in 2012 is the result of the analysis of several petabytes of data on a computing infrastructure with over 100,000 CPUs.

The emphasis of this thesis is placed on the meta-monitoring tool HappyFace and in particular its application to the GoeGrid cluster. Monitoring the services provided by the Grid infrastructure is essential for stable and performant operation. The focus is placed on services that provide job management at different levels. The objective of this thesis is to analyse two services, namely PBS and CREAM CE, which perform job management, and to provide a monitoring tool to generate detailed real-time monitoring information.

1.1. WLCG

The Worldwide LHC Computing Grid (WLCG) [9] provides the computing infrastructure for all four experiments of the LHC. This Grid infrastructure consists of over 150 computing centres in nearly 40 countries. Each site provides mass-storage and computing resources. The infrastructure is organised in a hierarchy, starting with the Tier-0 centre at CERN. Data from Tier 0 are replicated to the national Tier 1 centres and from there to the regional Tier 2 and local Tier 3 centres.

Tier 0: This tier consist of only the CERN Computer Centre [10] located on the CERN area. All data are preprocessed and permanently stored here as well as replicated to the Tier 1 sites.

Tier 1: These are the 12 large national sites. Each holds a partial copy of the Tier 0 data. They are responsible for reprocessing and distribution of data to Tier 2.

Tier 2: These mostly regional sites handle specific analysis tasks and a share of simulated event production and reconstruction. There are currently around 140 Tier 2 sites.

Tier 3 has no formal obligation to WLCG and consists of local clusters or single computers.

1.2. GoeGrid

GoeGrid is a Grid resource centre hosted at the GWDG (Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen) [11] and is mainly used as an ATLAS Tier 2 and Tier 3 centre in the WLCG. As part of the German Grid computing initiative D-Grid [12], GoeGrid also contributed to MediGrid [13] and TextGrid [14].

The GoeGrid cluster provides over 2500 CPU-cores as well as approximately 1 PB of storage. About 300 nodes are available. GoeGrid is running the Scientific Linux [15] distribution based on Red Hat Enterprise Linux [16].

As part of the WLCG, GoeGrid provides several services such as dCache [17] for mass storage or the BDII (Berkeley Database Information Index) [18] service for collecting and publishing of site status information. CREAM CE [19] is a front-end to the local batch system and handles the submission of Grid jobs.

Administrating a large Tier-2 cluster such as GoeGrid in the WLCG plays a key role to provide stably running Grid services and sufficient computing resources. In this thesis, two modules, namely the PBS module and CREAM module, are proposed and developed.

2. Grid and Job Scheduling System

2.1. Grid

Foster and Kesselman define a computing Grid as a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [20].

Foster published a list of three criteria which define a Grid infrastructure [21]:

- A Grid coordinates resources that are not subject to centralised control.
- A Grid uses standard, open, general-purpose protocols and interfaces.
- A Grid delivers nontrivial qualities of service.

A good example for a Grid is the WLCG, as its infrastructure spans several countries around the globe. It uses, amongst others, the open-source middleware gLite [22] and provides over 250 PB of storage.

An integral part of Grid operations is the management and distribution of jobs on the Grid. Jobs are self-contained and consist of data sets and applications that perform specific tasks on those sets. In the context of the WLCG, they perform tasks such as event reconstruction, calibration, simulation and analysis.

2.2. PBS

The Portable Batch System (PBS), specifically Terascale Open-Source Resource and QUEue Manager (TORQUE) [23], performs job scheduling on GoeGrid. It is used in conjunction with the Maui Cluster Scheduler [24]. Jobs can be submitted via the command

qsub with additional information like the required amount of memory, number of CPUs, estimated runtime or dependencies on other jobs. They are usually grouped in queues depending on their origin and properties and will be executed when a suitable node is available. For instance, jobs concerning the ATLAS experiment are organised in several queues. Likewise, all jobs that are expected to run for a long time can be managed in one queue. There are some other Local Resource Management Systems (LRMS), e.g. Load Sharing Facility (LSF) [25] or Oracle Grid Engine (OGE) [26], which perform similar tasks.

2.3. CREAM

The CREAM (Computing Resource Execution And Management) [19] service handles job submission and job management on GoeGrid while being fault-tolerant and providing user authentication. CREAM runs as a Java-Axis servlet on the Apache Tomcat [27] server.

CREAM runs on top of a Local Resource Management System (LRMS) and the BLAH (Batch Local Ascii Helper) service acts as an interface between CREAM and the underlying LRMS by providing a common interface for job management and job submission for all supported batch systems like PBS and LSF.

ICE (Interface to CREAM Environment) provides an interface between the WMS (Workload Management System) of the gLite middleware [22] and CREAM.

2.3.1. Job Life Cycle

As jobs are handed to the CREAM CE service, each job traverses a number of different states between submission and completion. Figure 2.1 shows the possible state transitions a job can undergo. Each job will start in the *registered* state, where it has been registered in the system but has not yet started running. It will then switch to the *pending* state, which means that the job is ready but has not yet been submitted to the LRMS. If successful, the job will now be scheduled by the LRMS and switch to the *idle* state. When suitable resources for execution are available, a wrapper, which will create an environment for the job, will be run and the state changes to *running*. Afterwards, the actual executable is started and the state changes to *really running*. Upon termination,

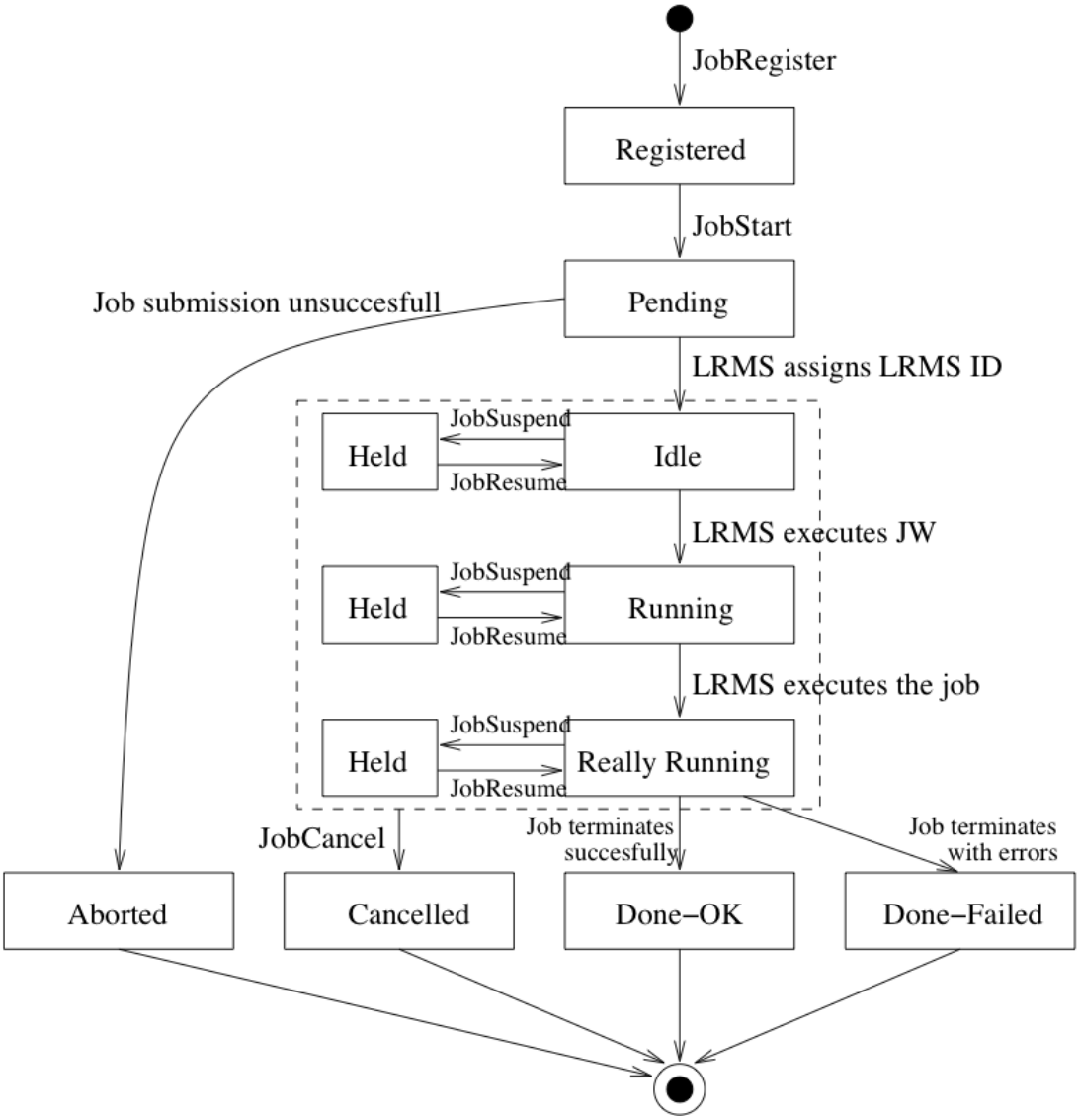


Figure 2.1.: Possible CREAM job state transitions [28].

the job will switch either to the *done-OK* or *done-failed* state, depending on whether any errors occurred during execution.

If the job submission to the LRMS fails, the state will be changed directly to *aborted*. While the job is either *idle*, *running*, or *really running*, it can be suspended to the *held* state and resumed later. When *idle*, *running*, or *really running*, the job can also be *cancelled* by the user.

2.3.2. JDL

The Job Description Language (JDL) is used to describe jobs that are submitted to the CREAM CE service [28]. A job description is composed by entries of the format

```
attribute=expression;
```

and each entry is terminated by the semicolon character. The job description has to be surrounded by square brackets. Comments can either be initiated using a sharp character "#" or a double slash "//". Comments spanning multiple lines can be encased between "/*" and "*/".

```
[
Type = "Job";
JobType = "Normal";
Executable = "myexe";
StdInput = "myinput.txt";
StdOutput = "message.txt";
StdError = "error.txt";
InputSandbox = {"/users/seredova/example/myinput.txt",
  "/users/seredova/example/myexe"};
OutputSandbox = {"message.txt", "error.txt"};
OutputSandboxBaseDestUri =
  "gsiftp://se.pd.infn.it/data/seredova";
]
```

While it is possible to include arbitrary attributes, only a certain set of attributes is recognised by CREAM CE. On the other hand, some of these attributes are mandatory and submitting a job without them will cause the request to fail.

Mandatory attributes include *Executable* or *QueueName*. Jobs may require a set of input files called the *InputSandbox* (ISB) and in turn store results in files in the *OutputSandbox* (OSB). The ISB is transferred to the worker node before execution starts and the OSB is retrieved after the job terminates.

2.3.3. Database

All job related information is stored in a central SQL database. This is described in more detail in Chapter 5.1.1.

3. The HappyFace Project

3.1. Monitoring

As a large infrastructure like GoeGrid consists of numerous systems and services, most of which are required for productive operation, the administration of this infrastructure is only possible if failures and their cause can be identified quickly.

Monitoring, meaning the surveillance of all involved components like computing hardware, infrastructure, software and services, is the general approach to this problem. Failures may be detected by analysing the collected data and the cause may be identified and rectified before it has serious effects.

Monitoring in the context of Grids uses several terms [29]:

Entity: Entities are unique networked resources like processors, memory, storage mediums, network links, applications and processes.

Event: Events are collections of time-stamped data associated with entities and a specific structure.

Event type: An event type uniquely identifies an event structure.

Event schema: An event schema defines the structure and semantics associated with an event type.

Sensor: A sensor generates events by monitoring an entity. There are *passive* sensors which collect already available data and *active* sensors which perform measurements themselves.

Monitoring in distributed systems like Grids generally contains four stages [29]:

Generation of events: Information is collected from entities and encoded according to a given schema.

Processing of events: Generated events are processed according to the specific application. This can include filtering or summarising and may take place at any stage of the monitoring process.

Distribution: Events are shared with any interested parties.

Presentation or consumption: The collected data are further processed to provide a visual presentation or are exported to a machine-readable representation.

There are multiple requirements for a general monitoring system [29]:

Scalability: A monitoring system should accommodate the growth of the infrastructure. Therefore it should achieve good performance to guarantee acceptable throughput and response time. On the other hand, it should not impact the performance of the monitored resources.

Extensibility: A monitoring system should be easily adjustable to changes in the monitored infrastructure. Hence, the underlying event handling has to be customisable and expandable without degrading the capacity of the monitored resources.

Data delivery models: A monitoring system should support different kinds of data delivery models to accommodate for static and dynamic events. In *push* models, the data transfer is initiated by the data source, while in *pull* models, the monitoring system actively queries the data source.

Portability: A monitoring system and particularly its sensors should be platform independent.

Security: Security services such as access control, single or mutual authentication, and secure transport of monitoring information are required in some scenarios and should consequently be supported by a monitoring system.

3.2. Meta-Monitoring

Monitoring can provide information about almost every part of a computing infrastructure. But this information is usually generated by a multitude of different

sources with different means of access. Administration would therefore require to check all sources for relevant information. Meta-monitoring aims to facilitate this process by aggregating monitoring data from these sources and presenting them in a user-friendly way.

As described in [30], a meta-monitoring system should meet several requirements:

Only one web site: All requested monitoring information should be accessible on one single website.

Up-to-date: The monitoring information should be renewed regularly, preferably close to real-time.

History functionality: Recorded monitoring data should be accessible via a history functionality. Accessing the history can help identifying correlations between problems reported by different monitoring sources at different times.

Fast access: A simple and optimised architecture can provide fast access to the monitoring information.

Comfortable: All monitoring information should be easily accessible.

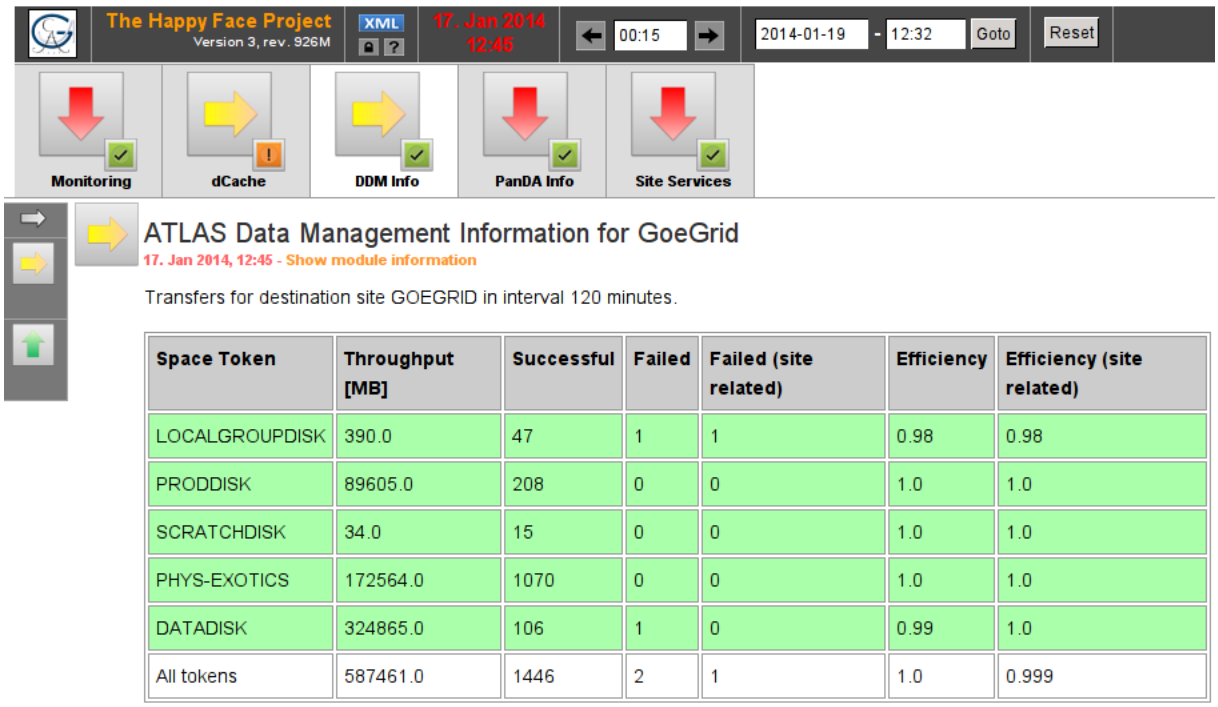
Simple warning system: Status information should be visualised in a simple and non-ambiguous way.

Customisable: The monitoring system should be adaptable to the monitored infrastructure.

3.3. The HappyFace Project

The HappyFace Project [31] aims to provide a meta-monitoring system that matches those requirements. Its purpose is to gather status information from existing monitoring sources and to create an overview for the whole site and for individual sources. It is a modular system consisting of a core and a number of modules specific to the site HappyFace is deployed on. Furthermore, data collection and visualisation are separated. Collected information is stored in the database and the history can later be accessed.

The core handles the execution of the modules including the generation of the individual web pages. It also provides a set of functions available to all modules, including access to



Space Token	Throughput [MB]	Successful	Failed	Failed (site related)	Efficiency	Efficiency (site related)
LOCALGROUPDISK	390.0	47	1	1	0.98	0.98
PRODDISK	89605.0	208	0	0	1.0	1.0
SCRATCHDISK	34.0	15	0	0	1.0	1.0
PHYS-EXOTICS	172564.0	1070	0	0	1.0	1.0
DATADISK	324865.0	106	1	0	0.99	1.0
All tokens	587461.0	1446	2	1	1.0	0.999

Figure 3.1.: The HappyFace web page.

the database. Modules on the HappyFace web page can be grouped into categories which will combine the output of the modules and derive a status value.

Each module is responsible for handling different data sources like log files, databases or web pages of another service. Each module provides separate functions for data acquisition and rendering. The acquisition function is called periodically and the results are stored in the database. Each time the web page is loaded, the stored information is fetched and passed to the template.

Figure 3.1 shows the HappyFace user interface. The bar at the top allows to access the history, listed below are the different categories with a status indicator. The left column shows the status of the individual modules in the selected category and permits to navigate to the module output quickly, which is shown next to it.

3.4. Versions

HappyFace version 3 is currently under active development and it has been deployed [32] at GoeGrid already. It is completely written in Python and uses HTML templates with embedded Python code for the user interface. Being a full rewrite of HappyFace version 2,

it aims to solve several structural issues present in the older version. HappyFace version 2 used PHP code embedded into the modules resulting in poor separation between data processing and rendering as well as increased complexity.

3.5. Installation

Besides Python 2 [33] of at least version 2.6, HappyFace requires the web framework CherryPy [34] version 3, the SQL toolkit and object relational mapper SQLAlchemy [35], the SQLAlchemy schema migration tools [36] and the template engine Mako [37]. The HappyFace core as well as the modules are maintained in separate SVN repositories. HappyFace can be downloaded from the central SVN repository by executing:

```
svn co https://ekptrac.physik.uni-karlsruhe.de/public/HappyFace/branches/v3.0 HappyFace
```

A set of modules can then be installed to the *modules/* subdirectory by running

```
svn co https://ekptrac.physik.uni-karlsruhe.de/public/HappyFaceModules/trunk modules
```

in the newly created *HappyFace/* directory. This repository currently contains nearly 50 modules including the modules that have been developed during this thesis.

3.6. Configuration

The subdirectories *defaultconfig/* and *config/* contain the configuration files of HappyFace and the modules. Files in *defaultconfig/* contain the default configuration values and should not be altered as they are under revision control and might change in future revisions. Instead, all values can be overridden by files in *config/*.

The subdirectory *modules-enabled/* contains the configuration for the individual module instances. By calling the *modconfig* tool, a basic configuration can be obtained:

```
python tools.py modconfig modulename
```

Modules can be grouped in categories which are configured in *categories-enabled/*. Each category has a name, an optional description and a list of the module instances. Additionally, an algorithm to calculate the category status from the module statuses may be specified.

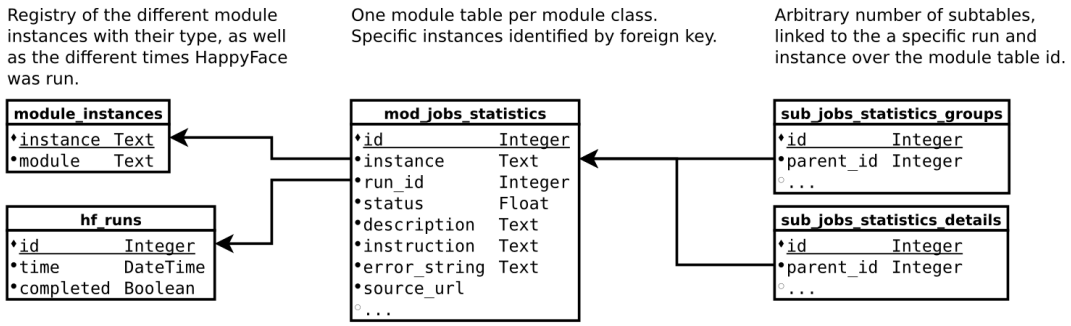


Figure 3.2.: HappyFace module tables [41].

3.7. Database

HappyFace uses SQLite [38] by default, which is a file-based database that does not require additional services or a setup. If this does not provide the necessary performance or needed features, it is also possible to connect HappyFace to a database server like MySQL [39] or PostgreSQL [40].

HappyFace stores all collected data from all modules in a database with layout shown in Figure 3.2. Each instance of each module is registered in the table *module_instances*. Every time HappyFace collects data, an entry is added to the table *hf_runs*, which stores the current date and time as well as a unique identifier for each run.

The table *mod_jobs_statistics* stores module specific information for each module instance on each run. As modules often require to store more than one data record per run, HappyFace supports additional subtables for each instance. The subtables allow the modules to store data in a module-specific layout and therefore provide great flexibility.

3.8. Modules

3.8.1. Module Structure

HappyFace modules are divided in two parts for data acquisition and rendering which are executed independently. The script *acquire.py* calls the function *prepareAcquisition* to get a list of the needed files. After fetching them, the function *extractData* is called

to extract the data from the files and store it in the database. If subtables are used, the function *fillSubtables* will get called to fill them with additional data.

The *render.py* script generates the HTML files visible on the web interface. The function *getTemplateData* can retrieve additional data from subtables. Each module has a corresponding HTML template file which can contain embedded Python code to display the data.

Each time a module is executed, one row is inserted into the module's table. An arbitrary number of fields can be added to the table, but for storing more complex relations, subtables should be used. In contrast to the module's table, a module can have any number of subtables and can insert more than one row per run.

3.8.2. Module Configuration

Module specific configuration is usually stored in *config/modules-enabled/*. Files are processed in alphabetical order, where each file generally contains one or more module instances:

```
[INSTANCE_NAME]
module = Plot
name =
description =
instruction =
type = rated
weight = 1.0
```

Each entry starts with the instance name encased in square brackets. The *module* attribute then specifies the module class. It is common to declare several module instances derived from the same module class which will run with different sets of parameters. Subsequently follows the verbose instance *name*, as it will be displayed on the HappyFace web page, and a *description*. Valid values for *type* are "rated", "unrated" and "plots". The status value of "rated" modules is included when determining the category status, while "unrated" modules are ignored. Modules with the *type* "plots" only indicate if they successfully collected data or not. The specified *weight* can be used by some algorithms to calculate the category status.

3.8.3. Category Configuration

Modules can then be grouped into categories with the configuration files in *config/categories-enabled/* which have a similar structure as the module configuration. Each entry is initiated with the category name in square brackets. The attributes are the verbose category *name*, a *description*, a *type* which is either "plots" or "rated" and an *algorithm* to calculate the status value. This can be either "worst" or "average".

3.8.4. Module Rating

Each module may return a status value indicating the health of the monitored resources. This value is then used by HappyFace to calculate a category status based on the specified algorithm. For normal module operation this is a float value in the interval $[0, 1]$, while it automatically gets assigned the value -1 if an error occurred during execution and -2 in case data retrieval failed. The interval is divided further into three states. A value in $[0.66, 1]$ corresponds to normal operation, if monitoring detects indication of problems, a value in $[0.33, 0.66)$ will be chosen. For serious errors, the status drops to a value in $[0, 0.33)$.

The category rating is derived from the module ratings of the modules contained in a category according to the specified algorithm. Modules with a rating type of *unrated* are not included in the calculation. The *worst* algorithm selects the lowest module status as category status. The *average* algorithm sets the category status to the average of all modules in the category, weighted with the value defined in the module configuration.

3.9. SQLAlchemy

SQLAlchemy [35] is an SQL toolkit and an ORM (object-relational mapper) for Python. SQLAlchemy supports several RDBMS (relational database management system) and also allows to map Python objects to database tables. As SQLAlchemy is already used internally for the HappyFace database, it was the obvious choice for interfacing with the CREAM database.

3.9.1. Reflection

One very useful feature of SQLAlchemy is called reflection. Given a database connection, it can retrieve the metadata of existing tables and construct the corresponding Python objects. It also fetches all tables referenced via foreign keys and retains all relations between tables, which makes complex queries spanning several tables easier, as most join conditions can be generated automatically. For example, the metadata of the *job* table from the CREAM database can simply be loaded by running:

```
job = Table('job', meta, autoload=True)
```

3.9.2. Backends

SQLAlchemy provides backends for several databases including MySQL, SQLite and PostgreSQL. These backends allow to use any of the supported databases without knowledge of the underlying interface by hiding differences and providing a uniform interface.

3.10. Mako

Mako [37] is a template engine for Python. It allows a clean separation between data processing as well as rendering and supersedes the usage of embedded PHP code in HappyFace 2.

Values from the main module table are passed to the template by default and it is possible to add or change data by overriding the *getTemplateData* function of the module. Single values might be inserted with `${expression}`, for instance

```
${module.dataset["status"]}
```

which will output the status value. Control structures such as *for*, *while* and *if* are also available and are written using the % marker.

```
% for job in jobs:
  <td>${job}</td>
% endfor
```

For even more flexibility, complete blocks of Python code can be inserted using `<%` and `%>`.

```
<%
for user in users:
  users[user].name = users[user].name.split('@')[0]
%>
```

It is also possible to define functions containing template code with `<%def>` and `</%def>`.

```
<%def name="print_row(list)">
<tr>
% for i in list:
  <td>${i}</td>
% endfor
</tr>
</%def>
```

They can then be used like any other expression.

```
<table>
% for t in table
  ${print_row(t)}
% endfor
</table>
```

4. SQL and MySQL

An RDBMS (relational database management system) is often the tool of choice to store and manage large amounts of data records and to access specific records as well as representing the relations between records. SQL (Structured Query Language) is used to interface with the RDBMS to store, alter or retrieve data. Examples for RDBMS which implement SQL are MySQL [39], MariaDB [42] and SQLite [38]. MariaDB is a fork of MySQL, which aims to provide full compatibility with MySQL. Other common systems are PostgreSQL [40], Oracle Database [43] and Microsoft SQL Server [44]. The following discussion will primarily cover MySQL and SQLite, as they are used for CREAM CE and HappyFace respectively, in the setup at GoeGrid. Nevertheless, most of it also applies to other RDBMS.

The central concept is the description of relations between attributes by using relational algebra. One kind of relation between attributes is the aggregation of attributes in a table. Each column contains the attribute values corresponding to the attribute. The rows in turn combine several attribute values and form data records. Moreover, it is desirable to reduce the redundancy of data stored in the database. For that purpose, data records are split over several tables and each table is expanded by an additional attribute, which makes each data record in the table uniquely identifiable. This attribute is called the primary key.

In order to realise a relation from one data record to another, the first record stores the key of the second record in an additional attribute. This is called a foreign key.

This allows to create one-to-one and many-to-one relations. Many-to-many relations require an additional table: by storing two keys for each one-to-one relation, an arbitrary amount of relations can be expressed.

4.1. Database Schemes

4.1.1. Data Types

Each column is assigned a data type which facilitates internal storage and defines the properties and allowed values.

Numeric MySQL provides several data types [45] for storing numbers. For storing integers, the INTEGER type is intended. This type is available in sizes from 8 to 64 bits and each has a SIGNED and an UNSIGNED variant, where SIGNED is the default.

type	size in bytes	unsigned range	signed range
TINYINT	1	0 to 2^8	$-(2^7)$ to 2^7-1
SMALLINT	2	0 to 2^{16}	$-(2^{15})$ to $2^{15}-1$
MEDIUMINT	3	0 to 2^{24}	$-(2^{23})$ to $2^{23}-1$
INT	4	0 to 2^{32}	$-(2^{31})$ to $2^{31}-1$
BIGINT	8	0 to 2^{64}	$-(2^{63})$ to $2^{63}-1$

The BOOLEAN type is a synonym for TINYINT where a value of zero is considered false and non-zero values are considered true.

The DECIMAL type allows the storage of fixed-point numbers. For floating-point numbers, the types FLOAT and DOUBLE are provided, where FLOAT is stored in single-precision (32 bits) and DOUBLE is stored in double-precision (64 bits).

Date and Time The DATE type allows the storage of dates in the range '1000-01-01' to '9999-12-31'. The DATETIME type extends DATE by a time in the range from '00:00:00' to '23:59:59' . The TIME type stores time in the range from '-838:59:59' to '838:59:59'.

A TIMESTAMP is in the range from '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC and is stored as the number of seconds since the UNIX epoch ('1970-01-01 00:00:00' UTC).

Strings For the storage of text, the types CHAR, TEXT and VARCHAR are available. TEXT has several variants of different maximum size, much like the integer types.

BINARY, VARBINARY and BLOB behave in much the same way, but do not respect a specific character encoding and can store any binary data.

4.1.2. Column Attributes

Furthermore, several attributes to change the behaviour of table columns are available.

Upon creation, one column per table can be declared as PRIMARY KEY, which then stores a unique identifier for each data record. The uniqueness is enforced by SQL and trying to insert duplicate values results in an error. This column could be a string storing names, but most common is the usage of an integer type with the AUTO_INCREMENT attribute. This attribute automatically assigns ascending numbers in the case that no value is provided. The NOT NULL constraint is also implied.

The UNIQUE constraint provides similar behaviour as the primary key, but can be defined on any columns and does not imply NOT NULL.

A FOREIGN KEY allows to reference the primary key of another table. Instead of just storing the value, this has the advantage that SQL enforces integrity. Only values that are present in the referenced table or NULL are considered valid and when a referenced entry is deleted, action is taken to preserve integrity. It can be specified whether the referencing entry should be deleted too or set to NULL.

The PRIMARY KEY and UNIQUE constraints have another very useful property. The values are indexed and stored in a B-tree[1], which allows for very fast retrieval of specific values. Without an index, finding all matching entries requires iterating over the whole table.

4.2. Queries

The most common operation in SQL is retrieving data via a query. This is accomplished with the SELECT keyword. The general structure of a query is

[1] A tree-like data structure that performs most operations in logarithmic time.

```
SELECT columns FROM tables WHERE condition GROUP BY column
  HAVING condition ORDER BY column LIMIT count;
```

Most of the keywords are optional and the parameters allow arbitrary complex constructs, even nesting queries is possible. On the other hand, queries can be as simple as

```
SELECT 6*7;
```

or the more useful

```
SELECT * FROM table;
```

to select all rows from the given table. The WHERE clause indicates the conditions that rows must satisfy to be selected. The GROUP BY clause groups rows according to the values in the specified columns and is mostly used in conjunction with aggregate functions. The HAVING clause imposes additional conditions on the selected values. In contrast to the WHERE clause, it can be used with aggregate functions. Similar to GROUP BY, the ORDER BY clause sorts according to the specified columns. The order can be modified with ASC for ascending or DESC for descending. Finally, the LIMIT clause restricts the number of selected rows to the given value.

Aggregate functions take a set of values and return a single value. They include:

COUNT returns the number of selected rows.

MIN selects the smallest value in the set.

MAX returns the largest value in the set.

AVG calculates the average value. NULL values are ignored.

SUM returns the sum of all given values. NULL is ignored.

4.2.1. Joins

As the desired values can often be distributed over several tables, *joins* allow queries to span multiple tables and combine the selected rows according to different rules.

INNER JOIN: An INNER JOIN without any additional conditions will return the Cartesian product of both tables. Otherwise, all combinations of rows that match the condition are returned.

LEFT JOIN: A LEFT JOIN is similar to INNER JOIN, but it returns all matching entries from the left table, even if no corresponding entries from the right table were found. The unavailable entries from the right table are filled with NULL.

RIGHT JOIN: A RIGHT JOIN is functionally equivalent to the LEFT JOIN, only the roles of the tables are reversed.

OUTER JOIN: An OUTER JOIN combines the behaviour of LEFT JOIN and RIGHT JOIN. All entries from both tables are returned. If two entries from both tables match, they are combined in a single row. Otherwise, the columns of the other table are filled with NULL.

5. New HappyFace Modules

Administrating a large system like the GoeGrid is facilitated by the availability of up-to-date information from all running services.

The PanDA (Production and Distributed Analysis) [46] system is used for job submission and distribution in the ATLAS Distributed Computing (ADC) [47] infrastructure, which is part of the WLCG.

While monitoring information from PanDA is available and accessible via its own HappyFace modules, this information is not always up-to-date and is specific to the PanDA service. It is not suitable to ascertain the health of the underlying services or monitor jobs that are submitted by other means.

With the framework provided by HappyFace, two modules were developed to address these shortcomings and to provide detailed real-time monitoring data. These modules are described in the following.

5.1. CREAM CE Module

The CREAM CE module was developed within this thesis with the goal of providing detailed information and statistics of all jobs managed by one CREAM instance. An overall summary is provided as well as more detailed statistics concerning queues, users and worker nodes. Figure 5.1 shows the queue summary.

5.1.1. Database Interface

To generate statistics, the CREAM module directly interfaces with the SQL database of the CREAM service. This database with the scheme shown in Figure 5.2 is usually called

Queue	REGISTERED	PENDING	IDLE	RUNNING	REALLY_RUNNING	CANCELLED	HELD	DONE_OK	DONE_FAILED	PURGED	ABORTED	Total
atlas	0	0	10	0	0	0	0	0	0	0	0	10
atlasL	0	0	7	0	0	0	0	0	0	0	0	7
atlasS	0	0	7	0	0	0	0	0	0	0	0	7
atlasXL	37	0	330	24	765	0	0	0	0	0	0	1156
ops	0	0	52	0	0	0	0	0	0	0	0	52
opsL	0	0	2	0	0	0	0	0	0	0	0	2
opsS	0	0	1	0	0	0	0	0	0	0	0	1
opsXL	0	0	3	0	0	0	0	0	0	0	0	3
Total	37	0	412	24	765	0	0	0	0	0	0	1238

Figure 5.1.: Queue summary as provided by the CREAM module.

creamdb. The relevant tables are *job*, *job_status* and *job_status_type_description*. The central table is the *job* table, where every job currently registered to CREAM is listed. Each job is identified by a unique *id* and other useful attributes are *queue* as well as *userId*, *localUser*, *workerNode* and *lrmsAbsLayerJobId*, which contains the job ID assigned by the LRMS.

The table *job_status* keeps track of the state of each registered job. Every time a job changes its state as described in Section 2.3.1, a new entry is added to the table. The corresponding job is identified by the *jobId* attribute while *timestamp* contains the time the change occurred and *type* denotes the new state the job switched to. Additionally, *exitCode* or *failureReason* are set in the case that the job finished or failed. The *name* attribute in *job_status_type_description* provides a human readable representation of the job status.

Upon database creation, the table *job_status_type_description* is initialised with the mapping between the status numbers from 0 to 10 and names [28]:

0. **REGISTERED:** the job has been registered (but not started yet).
1. **PENDING:** the job has been started, but it is still to be submitted to BLAH.
2. **IDLE:** the job is idling in the Local Resource Management System (LRMS).
3. **RUNNING:** the job wrapper which “encompasses” the user job is running in the LRMS.
4. **REALLY_RUNNING:** the actual user job (the one specified as Executable in the job JDL) is running in the LRMS.

5. **CANCELLED:** the job has been cancelled.

6. **HELD:** the job is held (suspended) in the LRMS.

7. **DONE_OK:** the job has successfully been executed.

8. **DONE_FAILED:** the job has been executed, but some errors occurred.

9. **PURGED:** the job has been purged.

10. **ABORTED:** errors occurred during the "management" of the job, e.g. the submission to the LRMS abstraction layer software (BLAH) failed.

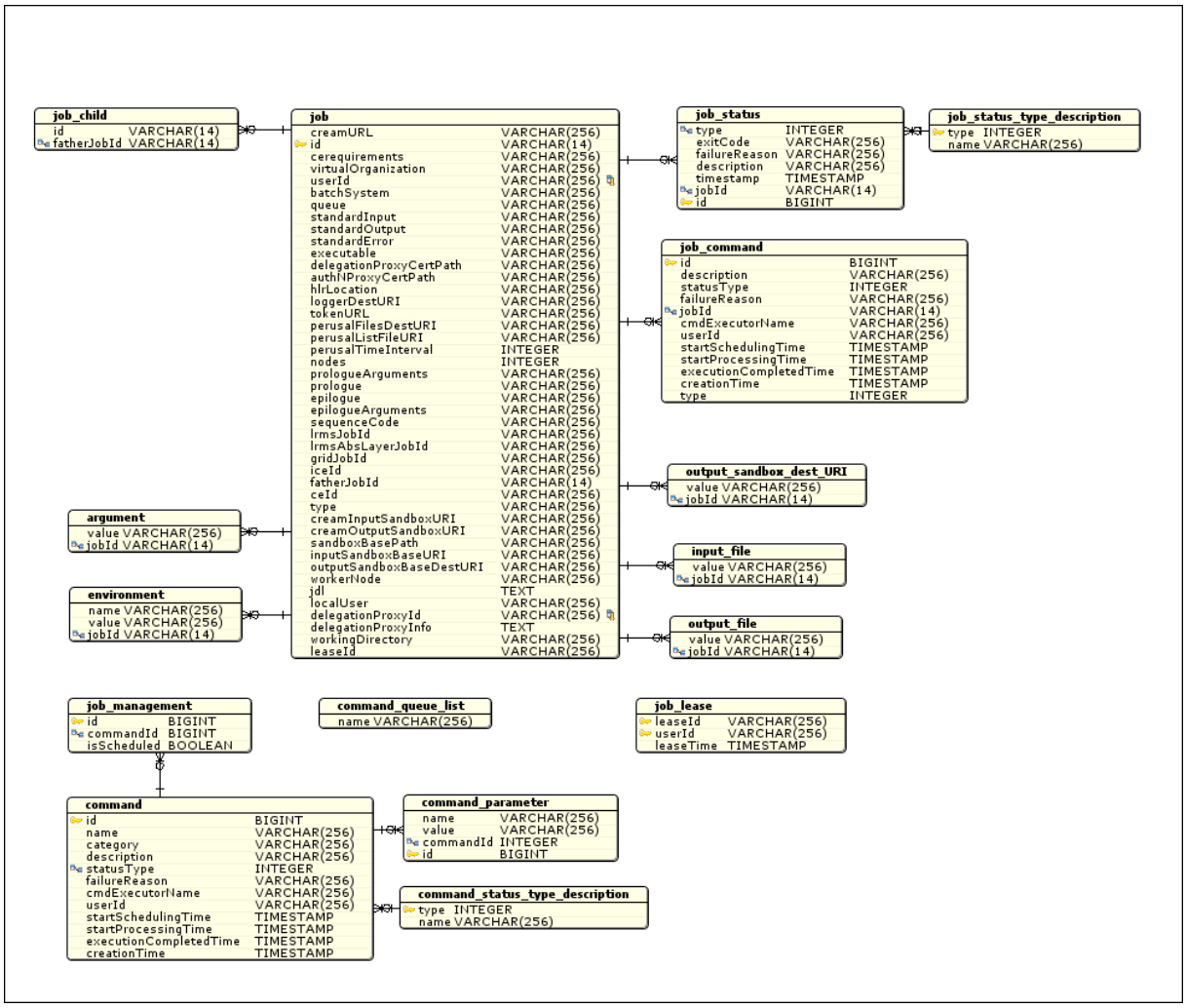


Figure 5.2.: CREAM database layout.

To get a list of all queues, the query

```
SELECT DISTINCT queue FROM job;
```

is used. Statistics for one queue can then be extracted by executing the query

```
SELECT type,count(*) FROM job_status WHERE id IN
  ( SELECT MAX(id) FROM job_status WHERE JobId IN
  (SELECT id FROM job WHERE queue=:q) GROUP BY JobId )
GROUP BY type WITH ROLLUP;
```

where *:q* is a placeholder for the queue name. This query first extracts all jobs assigned to the queue with the subquery:

```
SELECT id FROM job WHERE queue=:q
```

The result is then used to acquire the most recent *job_status* entry by means of the subquery:

```
SELECT MAX(id) FROM job_status WHERE JobId IN ( ... ) GROUP BY JobId
```

"MAX(id)" and "GROUP BY JobId" are employed to get the current *job_status* entry for each distinct *JobId*. As the *id* attribute is unique and has the auto increment property, the entry with the largest *id* for one *JobId* is guaranteed to be the most recent. The *timestamp* attribute on the other hand is not suitable, because it only has a second resolution but some state changes take less time and therefore have identical timestamps. Additionally, the timestamp is not always guaranteed to be exact and could vary when for example the system time is adjusted over NTP[1].

The outermost part

```
SELECT type,count(*) FROM job_status WHERE id IN ( ... )
GROUP BY type WITH ROLLUP;
```

then counts the number of occurrences for each type. In addition, "WITH ROLLUP" provides the total for all types.

This can then be repeated for *userId* and *workerNode* to gather additional statistics.

To calculate the total for all jobs, the query

```
SELECT type,count(*) FROM job_status WHERE id IN
  ( SELECT MAX(id) FROM job_status GROUP BY JobId )
GROUP BY type WITH ROLLUP;
```

is used.

This approach has several flaws:

[1] Network Time Protocol used for clock synchronisation.

- Even though only the most recent state is considered, it is counted regardless of the type. For most states this is the intended behaviour, but for CANCELLED, DONE_OK, DONE_FAILED, PURGED and ABORTED, the corresponding jobs are counted as long as the database entry exists. Those jobs are therefore counted until the entry is purged by CREAM, which causes finished jobs to "pile up" followed by a drop when the entries are removed.

- A large amount of queries is generated. Including the per-node statistics, over 300 queries would be issued each time. This could cause unnecessary load on the server.

To eliminate the first issue, job states are divided into two groups:

- The jobs that are still somehow active in CREAM: REGISTERED, PENDING, IDLE, RUNNING, REALLY_RUNNING, HELD

- Jobs that are no longer active and have reached one of the final states CANCELLED, DONE_OK, DONE_FAILED, PURGED, ABORTED

The first group can be handled as before while the second group requires an additional check whether the job finished in the last readout interval[2] or not. This check should be performed on the database server as the clock on the HappyFace server could differ. This would also require to first fetch the information for all affected jobs. While no job is counted too often with this approach, it is still theoretically possible to miss jobs in the case that entries are removed from the database before a readout happens. As this problem is caused by CREAM CE and should only affect a small fraction of jobs, this issue was not further investigated.

The second issue can be addressed by carefully rewriting the queries as it is possible to reduce the necessary amount to one query per summary. The queries are also adapted to take advantage of SQLAlchemy instead of using raw SQL. The queries in the following examples are the SQL output as generated by SQLAlchemy. The query addressing both issues is:

```
SELECT job.queue, job_status.type, count(*) AS count_1 FROM job
  INNER JOIN job_status ON job.id = job_status.`jobId`
  WHERE job_status.id IN (SELECT max(job_status.id) AS max_1
    FROM job_status GROUP BY job_status.`jobId`)
    AND (job_status.type IN  (0, 1, 2, 3, 4, 6)
    OR job_status.type NOT IN (0, 1, 2, 3, 4, 6)
    AND time_stamp > CURRENT_TIMESTAMP - INTERVAL 15 MINUTE)
GROUP BY  job.queue, job_status.type;
```

[2]This is 15 minutes for the default HappyFace configuration.

The subquery

```
SELECT max(job_status.id) AS max_1 FROM job_status
  GROUP BY job_status.`jobId`
```

is again used to select the most recent *job_status* entry for each job. The condition

```
  WHERE job_status.id IN (...)
    AND (job_status.type IN  (0, 1, 2, 3, 4, 6)
    OR job_status.type NOT IN (0, 1, 2, 3, 4, 6)
    AND time_stamp > CURRENT_TIMESTAMP - INTERVAL 15 MINUTE)
```

then selects only those who are either in an "active" state or have finished in the last 15 minutes. The integers correspond to the states REGISTERED, PENDING, IDLE, RUNNING, REALLY_RUNNING and HELD. The outermost part

```
SELECT job.queue, job_status.type, count(*) AS count_1 FROM job
  INNER JOIN job_status ON job.id = job_status.`jobId`
  WHERE ...
GROUP BY  job.queue, job_status.type;
```

then adds a column from the *job* table, in this case *queue*, and by grouping the *queue* and *type* entries generates the summary for all queues at once.

A similar problem concerning large amounts of queries arose while adding a detailed per-node job listing to the worker node summary. The first approach was to query the running jobs separately for each node. This resulted in very poor performance and about 300 queries on each readout. Further investigation revealed that most time was spent on selecting jobs corresponding to the specified node. As the CREAM database does not have an index for the *workerNode* attribute, this query required the server to check each entry of the *job* table which was then repeated about 300 times.

One possible solution would be adding an index for the *workerNode* attribute, which should speed up the queries considerably. This has the downside that a change to the CREAM database is required and neglecting to do so would at best result in an unusable module and could at worst substantially affect the CREAM service. Therefore, a different and less invasive approach was chosen: Retrieving the relevant jobs for all worker nodes is fast and can be done in only one query. The module then assigns the jobs to the nodes by iterating a single time over the list, which happens in linear time. This also avoided the same issue on the HappyFace database.

An unexpected performance issue was encountered when the module was moved from the development to the production system.

While the runtime of the queries was in the order of one-tenth second on the development system, each query took several minutes on the production system. At first it was believed

that this discrepancy was the result of a different MySQL configuration or a missing database index, but comparing the MySQL configurations did not lead to a conclusive result.

To investigate this problem, a virtual machine running Scientific Linux 6, the same distribution as used on the production system, was created on the development system.

The offending query

```
SELECT job.queue, job_status.type, count(*) AS count_1 FROM job
  INNER JOIN job_status ON job.id = job_status.`jobId`
  WHERE job_status.id IN (SELECT max(job_status.id) AS max_1
    FROM job_status GROUP BY job_status.`jobId`)
    AND (job_status.type IN  (0, 1, 2, 3, 4, 6)
    OR job_status.type NOT IN (0, 1, 2, 3, 4, 6)
    AND time_stamp > CURRENT_TIMESTAMP - INTERVAL 15 MINUTE)
GROUP BY  job.queue, job_status.type;
```

was incrementally shortened by removing all parts that did not have a negative effect on the performance. The resulting query was:

```
SELECT job_status.type FROM job_status WHERE job_status.id
  IN (SELECT max(job_status.id) FROM job_status
  GROUP BY job_status.jobId);
```

Applying the EXPLAIN statement to this query shows information about how it is executed. On the development system this leads to an output of

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<subquery2>	ALL	distinct_key	NULL	NULL	NULL	23357	
1	PRIMARY	job_status	eq_ref	PRIMARY	PRIMARY	8		1	
2	MATERIALIZED	job_status	index	NULL	fk_jobStatus_jobId_job_id	16	NULL	23357	Using index

while on the virtual machine, the output is:

id	select_type	table	type	possible_keys	key_len	ref	rows	Extra
1	PRIMARY	job_status	index	NULL	4	NULL	19442	Using where; Using index
2	DEPENDENT SUBQUERY	job_status	index	NULL	16	NULL	3	Using index

The *Extra* column reveals that an index is used for all parts of the query, so a missing index is probably not the cause. Very interesting on the other hand is the *select_type* value for the subquery. On the development system, the type is set to MATERIALIZED while on the virtual machine it is DEPENDENT SUBQUERY.

DEPENDENT SUBQUERY leads to the execution of the subquery for every element checked in the primary query and while the result of the subquery does not change during the execution of the primary query, the result is not reused but rather recalculated every time.

The MATERIALIZED subquery on the other hand is executed only once and the result is cached and reused for all rows checked by the primary query. This difference in behaviour can be explained by different versions of MySQL. While the development system uses MySQL 5.5, MySQL 5.1 is running on the virtual machine and the newer version offers some improvements in terms of query optimisation. Fortunately, as the issue is caused by the poor efficiency of the IN statement, the query can be rewritten to instead use the INNER JOIN statement:

```
SELECT job.queue, job_status.type, count(*) AS count_1 FROM
  (SELECT max(job_status.id) AS m
    FROM job_status GROUP BY job_status.`jobId`) AS a
  INNER JOIN job_status ON job_status.id = m
  INNER JOIN job ON job.id = job_status.`jobId`
    WHERE job_status.type IN (0, 1, 2, 3, 4, 6)
    OR job_status.type NOT IN (0, 1, 2, 3, 4, 6)
    AND time_stamp > CURRENT_TIMESTAMP - INTERVAL 15 MINUTE
GROUP BY job.queue, job_status.type;
```

This variant is over a thousand times faster on the virtual machine and shows no noticeable difference on the development machine.

5.1.2. Database Layout

The first version of the module stored data in a similar fashion it was later displayed on the web page. For the queue, user and node summary one database table for each is used.

This means that the tables have fixed width since each possible state is stored in a separate column. As each summary requires a new table, additional changes are necessary to add a new one. Since the type and layout of the stored information is almost identical in all cases, this was considered a superfluous complication. The fixed width was regarded as an unnecessary dependency on internal details of CREAM.

The layout was later rewritten to use a more general and flexible approach. The values are stored as a tuple consisting of *type*, *name*, *local_name*, *status* and *count*. *type* is either "queue", "user" or "node" and *name* stores the name of the corresponding queue, user or worker node. *local_name* is only set for the user summary and stores the user name

compute-0-8	0	0	1	0	0	0	0	0	0	0	0	1
compute-1-30.local	0	0	0	0	2	0	0	0	0	0	0	2
REALLY_RUNNING: CREAM950328077:pbs/20131209/17743776.pbs-goegrid.local:2013-12-09 14:57:15 CREAM797759577:pbs/20131209/17743799.pbs-goegrid.local:2013-12-09 15:00:53												
compute-1-31.local	0	0	0	0	4	0	0	0	0	0	0	4

Figure 5.3.: Node summary with detailed information for one node.

used on PBS while *name* stores the Grid user. *status* stores the CREAM status and *count* the number of jobs. As a result of the used CREAM database query, states with a count of zero are not reported and not stored in the HappyFace database, which reduces the amount of data further. This change allows to store all summaries in one table and permits easy expansion for new summaries.

5.1.3. HTML Template

The template contains the two helper functions *table* and *printRow*. The *table* function takes the arguments *title*, which is the title shown on top of the table and *list*, which is a list of all (*type*, *name*, *local_name*, *status*, *count*) tuples in the table. The other arguments are not used by default and may add additional information to the table. For the user summary, the *secondaryColumn* and *secondaryName* arguments are used to add an extra column for the local user. Finally, the *subtable* argument adds an additional row for each entry which is hidden by default and adds detailed job information to the node summary. This is shown in Figure 5.3.

5.1.4. Plotting

While the tables display detailed status information, they only cover the state of one readout interval and important values could simply be overlooked. It is therefore desirable to provide a quick overview by condensing the history over a longer period, preferably into a visual representation. For this purpose, HappyFace provides the Dynamic Plot Generator based on matplotlib that is capable of generating graphs of values stored in the HappyFace database. The plot generator is accessible via a set of functions as well as a web interface.

The plot generator takes a number of parameters which describe how the desired output should be generated. The desired time interval is specified with the *start_date*, *start_time*, *end_date* and *end_time* parameters. Furthermore, the *title* as well as the placement of the *legend* may be declared.

Curves are defined by specifying the module instance name, subtable and column. If the subtable is omitted, values are fetched from the main module table. Additionally, filters can be added to impose constraints on the allowed values. This is used to separate the curves by limiting each to a specific type.

A table storing all values for the plot generator is added. It contains the *type* and total *count* for all jobs.

5.2. PBS Module

Queue	Queued	Running	Held	Total
atlasL	29	27	0	56
atlasXL	133	1170	0	1303
dgiseq	3	0	0	3
elong	2	72	0	74
longtime	43	20	0	63
norun	0	1	0	1
shorttime	13	213	0	226
Total	223	1503	0	1726

Figure 5.4.: Queue summary as provided by the PBS module.

The first version of the PBS module was developed during the *Spezialisierungspraktikum*. The module provides functionality similar to the CREAM module, but instead of interfacing with the CREAM database, it uses data provided by the underlying PBS service in the form of a log file. Compared to the CREAM module, it does not provide any information of the Grid user who submitted a job, but on the other hand shows jobs that were not submitted through the Grid infrastructure but directly to PBS. Figure 5.4 shows a part of the module web page.

5.2.1. Qstat File Format

PBS provides information about the jobs currently registered to the system in the form of a log file. This log file contains a list of all jobs as well as detailed information for each job. An abbreviated entry for one job looks like this:

```
Job Id: 16800068.pbs-goegrid.local
    Job_Name = crea2_675925389
    Job_Owner = atplt010@creamce2.local
    resources_used.cput = 00:00:27
    resources_used.mem = 119492kb
    resources_used.vmem = 541532kb
    resources_used.walltime = 09:33:24
    job_state = R
    queue = atlasXL
```

Each entry starts with a unique job ID followed by the job attributes of the form *attribute_name = value*. An entry usually takes one line, but longer entries can be split across several lines, each indented by one tabulator. *extractData* parses the log file and stores all jobs and their attributes in a Python dictionary for further processing. Of all the present attributes, only a few are actually evaluated.

The field *job_state* contains a one-character code to indicate the job state:

Q – The job is queued and will run when the resources are available.

R – The job is running.

H – The job is held and will stay in that state until released or cancelled.

The field *queue* indicates to which queue the job belongs. This is used to generate a per-queue summary. The user name is extracted by splitting the *Job_Owner* field. This is used to generate a per-user summary.

5.2.2. Adaption of CREAM Module

While the development of the PBS module started before the CREAM module, both modules are similar in functionality. The PBS module was therefore later adapted to use a similar database layout as the CREAM module. This allowed to share code between both modules and made rendering and plotting available for both modules.

6. Estimation

Figure 6.1 shows the data collected so far. The values fluctuate, but the biggest contributions in normal operation over the observed period are the *really_running* jobs, followed by the *idle* jobs. A nearly constant amount of *running* jobs is also present. Most jobs finish as *done_ok* while some are occasionally *cancelled* or *done_failed*.

81203 jobs reached a final state. The following table shows a breakdown of the involved states.

State	Count	Fraction
CANCELLED	2077	2.56%
DONE_OK	61596	75.85%
DONE_FAILED	5478	6.75%
PURGED	0	0.00%
ABORTED	12052	14.84%

Most jobs reached the *done_ok* state, as would be expected for normal operation. About 15% of the jobs were *aborted*, most of them during an outage. About 7% reached the *done_failed* state and nearly 3% were *cancelled*, which can be considered normal.

Two outages were recorded during the observation. The short Python script in Listing A.1 was used to create the plots from the HappyFace database.

The first outage occurred around the 17th of January, as can be seen in Figure 6.2. The increasing number of *aborted* jobs is the first indication of the failure, reaching over 50 jobs in 15 minutes and peaking at about 150. *Aborted* jobs denote errors during job management and as it turned out, the failure was indeed caused by a misconfiguration of PBS, which did no longer accept submitted jobs. This is accompanied by a rising amount of *registered* and *pending* jobs and a decreasing amount of *done_ok* jobs. The number of *idle* and *really_running* jobs steadily drops. This is not surprising as PBS no longer spawns new jobs. Those jobs then stay *registered* or *pending* or are *aborted*. As

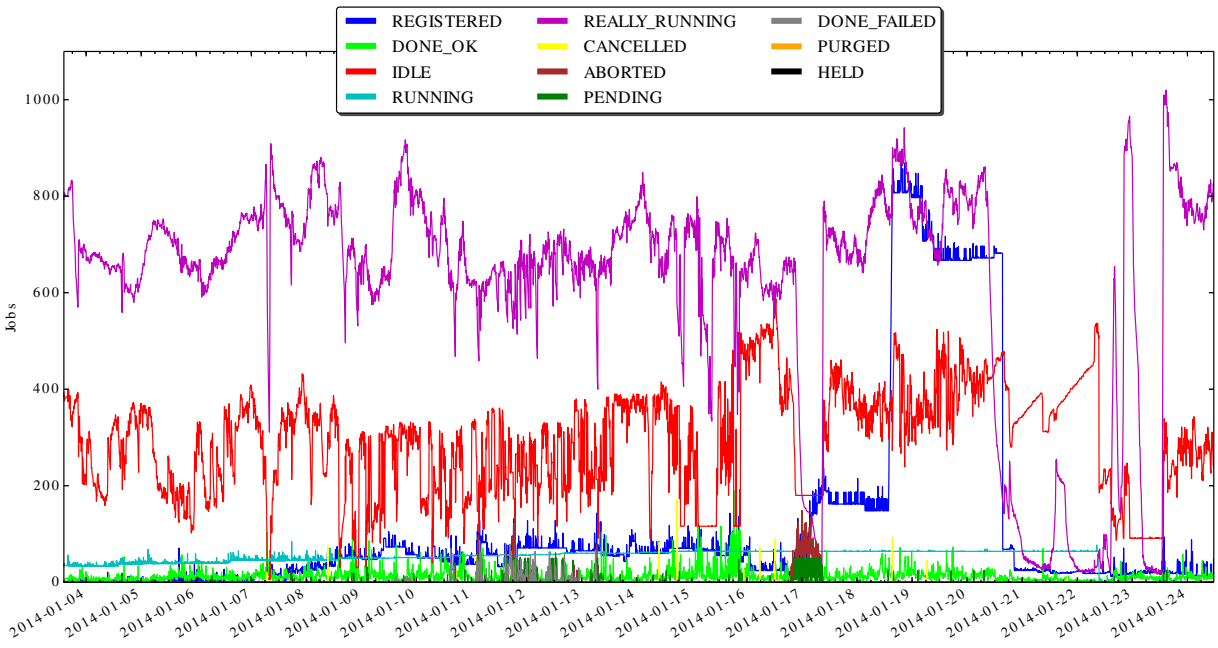


Figure 6.1.: Monitoring data collected by the CREAM CE HappyFace module for a specific CREAM instance.

the amount of *really_running* jobs decreases, less jobs reach the *done_ok* state. After the issue was resolved, the values quickly returned to their original amount with only the *registered* jobs staying at an elevated level.

Another outage happened between the 20th and 24th of January and is shown in Figure 6.3. Notable is a sudden jump in the number of *registered* jobs between the 18th and 19th, peaking at about 800, even overtaking the amount of *really_running* jobs.

The amount of *registered* jobs then suddenly drops two days later. As there is no increase of any other job states at this time and both points are exactly two days apart, it is safe to assume that those jobs triggered a timeout and were purged from the database by CREAM. The number of *really_running* jobs also drops similar to the first outage, but the amount of *idle* jobs remains relatively constant and then drops about two days later. Another difference is the absence of *aborted* jobs and a sudden reduction of *running* jobs. The *done_ok* jobs also diminish slightly.

During this time, the submission ob jobs from CREAM to PBS failed frequently. The exact cause is unknown, but restarting PBS rectified the issue.

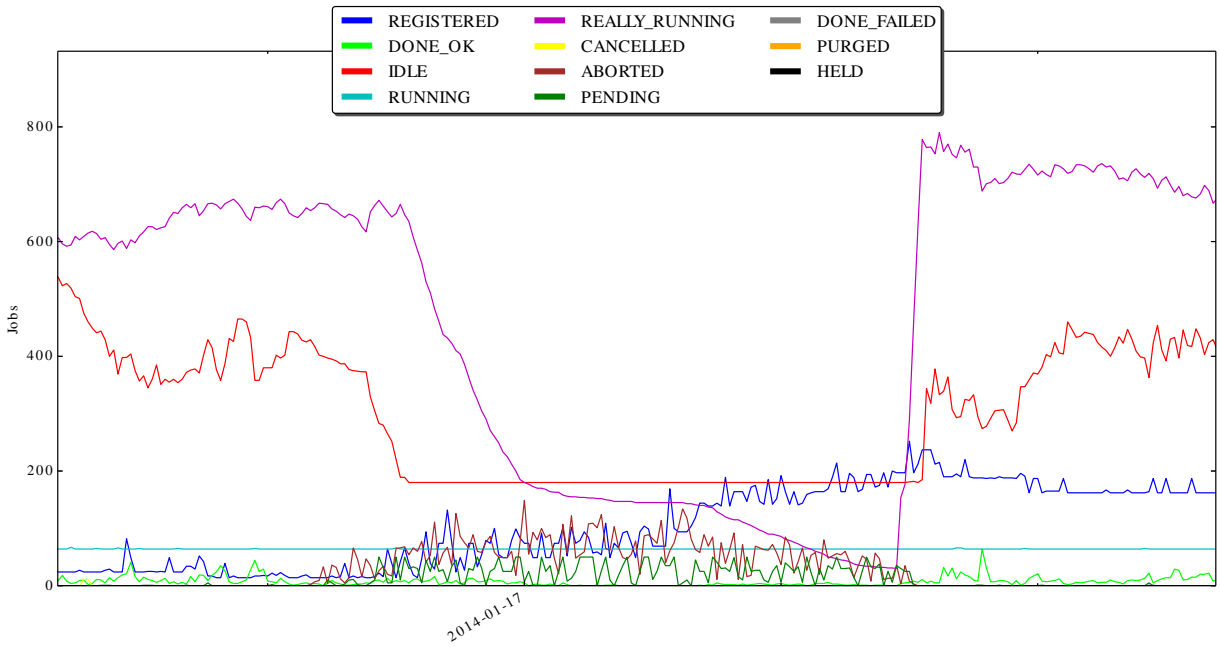


Figure 6.2.: Collected data during an outage around the 17th of January.

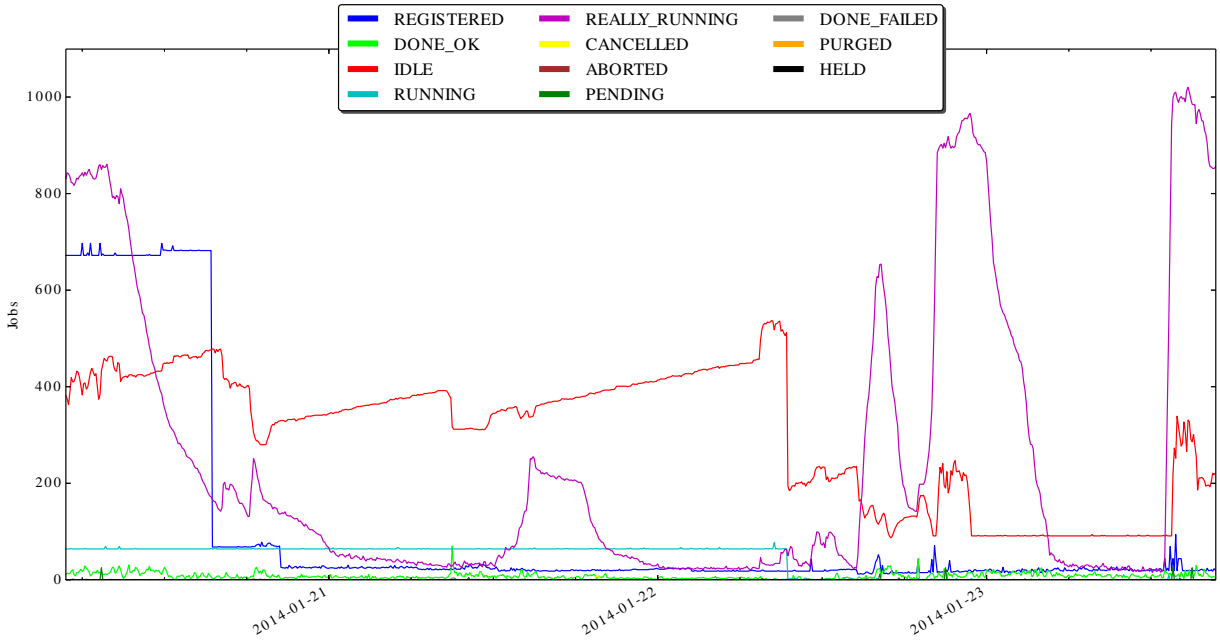


Figure 6.3.: Collected data during an outage between the 20th and 24th of January.

7. Conclusion and Outlook

7.1. Conclusion

The developed modules successfully provide detailed real-time monitoring of PBS and CREAM CE, therefore meeting the set goal. Both modules produce general statistics which allow to quickly assert the overall health of the monitored services as well as more detailed information that may help to narrow down the cause of a failure. This is further facilitated by providing a visualisation of the recorded history.

The inclusion of both modules in the official repository allows other Grid sites apart from GoeGrid to benefit from the HappyFace development. Additionally, as more modules become available, the attractiveness of HappyFace for other computer centres increases, possibly leading to the adaption of HappyFace as meta-monitoring tool.

7.2. Outlook

The modules presently provide solely informational functionality, meaning that they collect and visualise monitoring information but do not derive a rating. As described in Chapter 6 there are several quantities which could indicate errors, but further observation and research is necessary to better understand correlations between the values and to identify different kinds of failures.

The CREAM module currently interfaces directly with the database. The access to the database is restricted to read-only, but it is not desirable or even possible to expose the database for monitoring purposes, as some of the exposed information could lead to security issues. This could hinder the adaption of this module at other sites.

It would therefore be desirable to create an additional layer between the database and the module. This layer could then aggregate monitoring-relevant information from the

database and remove security-relevant and unnecessary information. By using a suitable protocol, this layer could also handle user-authentication and remote access.

Both modules use the HappyFace plot generator for data visualisation and generate an overview over the last 24 hours. While the plot generator is suitable for this use case, it is not very versatile and does not provide many options to customise the output.

The direct use of matplotlib or a similar library would allow to provide a more informative and appealing visualisation. As the number of jobs per time interval reaching a final state is very low and fluctuating, readability could be improved by integrating over the whole plotted interval, which is not feasible with the plot generator.

A. Appendix

```
1  from sqlalchemy import *
2  import datetime
3  import matplotlib.pyplot as plt
4  from matplotlib.dates import DayLocator, HourLocator, DateFormatter, drange
5  from numpy import arange
6
7  engine = create_engine('sqlite:///HappyFace.db')
8  meta = MetaData()
9  meta.bind = engine
10 hf_runs = Table('hf_runs', meta, autoload=True)
11 sub_creamce_status = Table('sub_creamce_status', meta, autoload=True)
12
13 res = select([hf_runs.c.time, sub_creamce_status.c.status, func.sum(sub_creamce_status.c.
       count)]).group_by(sub_creamce_status.c.parent_id).group_by(sub_creamce_status.c.
       status).select_from(hf_runs.join(sub_creamce_status,hf_runs.c.id==sub_creamce_status.
       c.parent_id)).where(sub_creamce_status.c.type=='queue').execute()
14
15 t = []
16 s = [[],[],[],[],[],[],[],[],[],[],[]]
17
18 currdate = None
19
20 for i in res:
21         if i[0]!=currdate:
22                 currdate = i[0]
23                 t.append(currdate)
24                 for x in range(0,11):
25                         s[x].append(0)
26
27         s[i[1]][-1]=i[2]
28
29 fig, ax = plt.subplots()
30
31 colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'lime', 'purple', 'orangered', 'brown']
32
33 for x in range(0,11):
34         ax.plot_date(t, s[x], '-',color=colors[x])
35
36 legend = ['REGISTERED','PENDING','IDLE','RUNNING','REALLY_RUNNING','CANCELLED','HELD','
       DONE_OK','DONE_FAILED','PURGED','ABORTED']
37
38 ax.legend(legend,loc='upper center', bbox_to_anchor=(0.5, 1.1), ncol=3, fancybox=True,
       shadow=True)
39
```

```
40 ax.set_xlim( t[0], t[-1] )
41 ax.set_ylim(0,1100)
42
43 ax.xaxis.set_major_locator( DayLocator() )
44 ax.xaxis.set_minor_locator( HourLocator(arange(0,25,6)) )
45 ax.xaxis.set_major_formatter( DateFormatter('%Y-%m-%d') )
46
47 ax.set_ylabel('Jobs')
48
49 ax.fmt_xdata = DateFormatter('%Y-%m-%d %H:%M:%S')
50 fig.autofmt_xdate()
51
52 plt.show()
```

Listing A.1: Python script to visualise monitoring data.

Bibliography

[1] Large Hadron Collider (LHC), URL http://home.web.cern.ch/about/accelerators/large-hadron-collider

[2] European Organization for Nuclear Research (CERN), URL http://www.cern.ch/

[3] A Large Ion Collider Experiment (ALICE), URL http://aliceinfo.cern.ch/

[4] A Toroidal LHC ApparatuS Collaboration (ATLAS), URL http://atlas.web.cern.ch/Atlas/Collaboration/

[5] Compact Muon Solenoid (CMS) Experiment, URL http://cms.web.cern.ch/

[6] Large Hadron Collider beauty (LHCb), URL http://lhcb.web.cern.ch/lhcb/

[7] G. Aad, et al. (ATLAS Collaboration), *Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC*, Phys.Lett. **B716**, 1 (2012), 1207.7214

[8] S. Chatrchyan, et al. (CMS Collaboration), *Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC*, Phys.Lett. **B716**, 30 (2012), 1207.7235

[9] Worldwide LHC Computing Grid (WLCG), URL http://wlcg.web.cern.ch/

[10] *The CERN Data Centre*, URL http://information-technology.web.cern.ch/about/computer-centre

[11] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG), URL http://www.gwdg.de/

[12] *D-Grid*, URL http://www.d-grid.de/

[13] MediGRID, URL http://www.medigrid.de/

[14] TextGRID, URL `http://www.textgrid.de/`

[15] *Scientific Linux*, URL `https://www.scientificlinux.org/`

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

[17] M. Ernst, P. Fuhrmann, M. Gasthuber, T. Mkrtchyan, C. Waldman, *dCache, a distributed storage data caching system*, CHEP (2001)

[18] L. Field, M. Schulz, *Grid deployment experiences: The path to a production quality LDAP based grid information system* pages 723–726 (2005)

[19] P. Andreetto, et al., *CREAM: A simple, Grid-accessible, Job Management System for local Computational Resources*, CHEP (2006)

[20] I. Foster, C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)

[21] I. Foster, *What is the Grid? - a three point checklist*, GRIDtoday **1(6)** (2002)

[22] E. Laure, F. Hemmer, F. Prelz, S. Beco, S. Fisher, M. Livny, L. Guy, M. Barroso, P. Buncic, P. Z. Kunszt, A. Di Meglio, A. Aimar, A. Edlund, D. Groep, F. Pacini, M. Sgaravatto, O. Mulmo, *Middleware for the next generation Grid infrastructure* **(EGEE-PUB-2004-002)**, 4 p (2004)

[23] *Torque Batch System*, URL `http://www.adaptivecomputing.com/products/open-source/torque/`

[24] *Maui Batch Scheduler*, URL `http://www.adaptivecomputing.com/products/open-source/maui/`

[25] *IBM Platform LSF*, URL `http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/`

[26] *Oracle Grid Engine*, URL `http://www.oracle.com/technetwork/oem/grid-engine-166852.html`

[27] *Apache Tomcat*, URL `http://tomcat.apache.org/`

[28] *CREAM User's Guide*, URL `https://wiki.italiangrid.it/twiki/bin/view/CREAM/UserGuideEMI3`

[29] S. Zanikolas, R. Sakellariou, *A Taxonomy of Grid Monitoring Systems*, Future Gener. Comput. Syst. **21(1)**, 163 (2005)

[30] 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(6)**, 062057 (2010)

[31] HappyFace, URL `https://ekptrac.physik.uni-karlsruhe.de/trac/HappyFace`

[32] *HappyFace instance for GoeGrid*, URL `http://happyface-goegrid.gwdg.de/`

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

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

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

[36] *SQLAlchemy Schema Migration Tools*, URL `https://code.google.com/p/sqlalchemy-migrate/`

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

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

[39] *MySQL Database*, URL `http://www.mysql.com/`

[40] *PostgreSQL Database*, URL `http://www.postgresql.org/`

[41] *HappyFace 3.0 RC1 documentation*, URL `http://ekphappyface.physik.uni-karlsruhe.de/~happyface/docs/index.html`

[42] *MariaDB*, URL `https://mariadb.org/`

[43] *Oracle Database*, URL `http://www.oracle.com/us/products/database/overview/index.html`

[44] *Microsoft SQL Server*, URL `http://www.microsoft.com/en-us/sqlserver/default.aspx`

[45] *MySQL Reference Manual*, URL `http://dev.mysql.com/doc/en/`

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

[47] S. Jézéquel, G. Stewart, *ATLAS Distributed Computing Operations: Experience and improvements after 2 full years of data-taking*, Journal of Physics: Conference Series **396(3)**, 032058 (2012)

Danksagung

Zunächst möchte ich Prof. Dr. Arnulf Quadt danken, der mir die Möglichkeit gegeben hat an diesem interessanten Thema zu arbeiten. Außerdem möchte ich ihm und meinem Zweitgutacher Priv.Doz. Dr. Jörn Große-Knetter für die Zeit und Mühe danken, die sie für meine Bachelorarbeit aufwenden.

Weiterhin gilt mein Dank Dr. Gen Kawamura, Erekle Magradze, Haykuhi Musheghyan und Dr. Jordi Nadal für ihre Unterstützung und ihren Rat sowie für die freundliche Arbeitsatmosphäre.

Schließlich danke ich Christian Wehrberger für seine entscheidene Unterstützung zu Beginn meiner Arbeit sowie Nikolai Wyderka, Robert Czechowski, Martin Ochmann, Janjenka Szillat, Johannes Frey und Christopher Eckner.

Erklärung nach §13(8) 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 11. Februar 2014

(Eric Buschmann)