# Development of a Plugin for Transporting Jobs Through a Grid

Jesper Koivumäki

Arcada – Nylands Svenska Yrkeshögskola

Helsingfors 2010

| **EXAMENSARBETE** | |
|---|---|
| Arcada | |
| | |
| Utbildningsprogram: | Informationsteknik |
| | |
| Identifikationsnummer: | 2796 |
| Författare: | Jesper Koivumäki |
| Arbetets namn: | Utveckling av en plugin-modul för transport av programkörningar genom ett beräkningsnät |
| Handledare (Arcada): | Göran Pulkkis |
| | |
| Uppdragsgivare: | Helsingfors Institut för Fysik, Teknologi Programmet |
| | |


Sammandrag:

Sensorn Compact Muon Solenoid (CMS) vid partikelacceleratorn Large Hadron Collider (LHC) hos Europeiska Organisationen för Kärnforskning (CERN) kommer att producera stora mängder data som måste analyseras med hjälp av datorresurser över hela Europa. För att underlätta distribueringen av datat har man utvecklat en typ av beräkningsnät kallat ett grid. Helsingfors Fysikinstitut (HIP) har gått med på att aktivt bidra till projektet genom att skänka beräkningsresurser till gridet. Beräkningsresurserna består främst av beräkningselement (CE) och datalagringselement (SE). För att möjliggöra användningen av dessa resurser programmerades en plugin-modul för att CMS Remote Analysis Builder (CRAB) skall kunna skicka beräkningsjobb till beräkningsresurserna via Advanced Resource Connector (ARC) Grid Middleware. Detta slutarbete beskriver plugin-modulen och dess egenskaper samt de gridkomponenter som är relevanta för plugin-modulens realisering.



| | |
|---|---|
| Nyckelord: | Grid, ARC, middleware, grid job, LHC, CMS, high-performance computing |
| Sidantal: | 38 |
| Språk: | Engelska |
| Datum för godkännande: | 9.4 2010 |

| **DEGREE THESIS** | |
|---|---|
| Arcada | |
| | |
| Degree Programme: | Information Technology |
| | |
| Identification number: | 2796 |
| Author: | Jesper Koivumäki |
| Title: | Development of a Plugin for Transporting Jobs Through a Grid |
| Supervisor (Arcada): | Göran Pulkkis |
| | |
| Commissioned by: | The Helsinki Institute of Physics, Technology Programme |


Abstract:

The Compact Muon Solenoid (CMS) sensor at the particle accelerator called the Large Hadron Collider (LHC) in the European Organizaion for Nuclear Research (CERN) will produce large amounts of data that will require computing resources all over Europe for analysis. In order to facilitate the distribution of data a distributed network called a grid was developed. The Helsinki Institute for Physics (HIP) has pledged to actively participate in this project by supplying computing resources. The computational resources mainly consist of computing elements (CE) and storage elements (SE). To make the resources available, a plug-in was programmed to enable the CMS Remote Analysis Builder (CRAB) to use the Advanced Resource Connector (ARC) Grid Middleware for submitting jobs to these computational resources. This thesis describes the plug-in as well as the grid components which are relevant to implementation of the plug-in.



| Keywords: | Grid, ARC, middleware, grid job, LHC, CMS, high-performance computing |
|---|---|
| Number of pages: | 38 |
| Language: | English |
| Date of acceptance: | 9.4 2010 |

# CONTENTS

# Figures

# 1 INTRODUCTION

As computers are becoming more and more powerful scientists are using this to their advantage. Moving trial and error into the virtual world has granted the opportunity to significantly increase the rate of testing by ways of simulation. Simulating events in the micro or macro scale, within a reasonable timeframe, requires a high level of detail and enormous amounts of computing power. Exploiting the advances in technology as means to advance researching methods is sometimes referred to as e-science.

Even as the processing power of modern computers increases steadily, the demand for more resources grows. One way of dealing with this has been the combining and sharing of resources. One technology that allows for this is often referred to as Grid Technology. Grids allow different organisations around the world to share their resources in a managed way, while still remaining in control over their own resources.

One of the most resource demanding projects in the world today, in regards to computational resources, is the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN). A lot of work has been done on solving the LHC compatibility problems in order to prepare for its completion since the work on the actual collider began. In 2001, a collaboration called NorduGrid was founded aiming to build a grid infrastructure in the Nordic countries suitable for production-level research tasks. One of the ways this was accomplished was by writing a job handling middleware, NorduGrid middleware or Advanced Resource Connector (ARC) as it's now called.

The LHC project at CERN is divided into numerous subprojects. The collider will speed up particles in order to let them collide and then measured data is extracted from these collisions. There are four main detectors set up around the LHC (CERN, 2009) with the sole purpose of measuring the data from the collisions; A Large Ion Collider Experiment (ALICE), A Toroidal LHC Apparatus (ATLAS), the Compact Muon Solenoid (CMS) (CERN, 2009) and LHC Beauty (LHCb). These four detectors are all represented by different teams with individual projects, all focusing on slightly different aspects of the collisions.

Each detector consists of a large number of sensors. The sensors will produce an enormous amount of data. This data will be stored in a tiered storage solution where it will be processed and filtered as it trickles down from the raw data storage at Tier-0 all the way to the end-user level Tier-2. The latter tier is the one where the analysed data is stored.

The CMS analysis software framework is called CMSSW and used for user analysis jobs. The jobs are sent by a user to a computational resource, using a software, mainly written in Python, called CMS Remote Analysis Builder (CRAB), through a virtual layer consisting of authorisation, resource brokering, queuing and state checking. In order for this to work, the different parts need to fit together. This can be quite a challenge, even if a lot of the software has been written especially to solve the computational problems regarding the LHC.

The Helsinki Institute of Physics (HIP) is a physics research institute jointly operated by five separate universities in Finland. HIP is involved in the research regarding the CMS and the data that sensor will generate during the collisions. Therefore a part of the work preparing for the time when the LHC is fully operational means making sure the analyses and data transfers function smoothly. At HIP, the task of preparing the resources is handled by the Technology Programme.

Because the academic computing resources in Finland use ARC middleware while the CERN resources use gLite Middleware, jobs cannot be sent directly from CERN to Finland. The task assigned to the Technology Programme by HIP was to build a plugin for CRAB enabling the sending of CRAB jobs through the ARC middleware. The plugin will then be proposed to be a part of the CRAB package for the benefit and ease of the users, enabling them to send jobs directly to the Finnish resources. The idea was to eventually use ARCLIB, a library of Python methods supplied with ARC, and also include this library as a part of the actual plug-in in the full CRAB package, but this cannot yet be done because of certain conflicts regarding Python environments.

Since the group of plugins called schedulers handle all user interaction, the new plugin meant to implement ARC compatibility was to be transparent to the end user. This meant that, from a user point of view, it had to behave exactly like the other schedulers. The new scheduler was also not to change the look and feel of the configuration files used for jobs assigned to be sent using ARC.

In Chapter 2, I will briefly explain what a grid is and go into detail regarding how the scientific grid used at CERN works, including the security layer that makes up a significant part of it. Chapter 3 describes ARC, how it’s used and what it actually is. Chapter 4 describes the software needed to perform CMS analyses. Chapter 5 describes the parts of CRAB that handle the communication with ARC. This is the main theme in this thesis. Chapter 6 contains the conclusions.

## 1.1 Participation

In February 2008 it became clear that there would be conflicts between the gLite and ARC middleware that would counteract Finland's participation in the CMS sensor research. At this time I was working for the Helsinki Institute of Physics. I was asked to study CRAB and participate in the designing and coding of the plugin that became the subject of this thesis and would enable submitting jobs over the ARC middleware.

The coding started during the spring in 2008 and in September 2009 the plugin was taken into production use for thorough testing. During this time I was joined by Eric Edelmann from the Finnish CSC (IT Centre for Science) who had already written a lot of Python scripts for other parts of CRAB. Eric rewrote a significant part of the existing code and was of significant help in getting the plugin to production use.

# 2 GRIDS

"What is there in common between the fight against avian flu, the development of drugs against malaria, the quest to understand the first instants of the Universe and research on climate change? All these topics require a huge amount of computer power and data storage capacity that can be satisfied with the Grid." (CERN, 2009)

In July 2002, Ian Foster wrote a paper (Foster, 2002) explaining the need for a definition of the concept "grid". He also proposed the following definition: "[The Grid is a system that] coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of service." This definition is still used today and is applicable to all scientific grids mentioned in this thesis.

As a more practical approach to defining a grid, one can consider "The 5 Big Ideas" that CERN has presented. (Grey, 2009) These five points describe the five ideas which drive the design of the grids used in CERN's research: global sharing of resources, a mutual trust between users and resources, an efficient and balanced use of resources, negligibility of distance between users and resources and open standards, so that combining grids is a possibility.

## 2.1 Types of Grids

Grids are basically the next step from supercomputers and clusters. A grid uses the Internet as a medium for connecting resources around the world and presenting them as one single resource to the user. These resources can be defined as a number of things, but are mainly either computing or storage resources. The grids relevant to this thesis consist of both computing and storage resources, but are considered as computing grids.

Grids come in various forms, depending on the purpose and user base. When talking about scientific grids, which are aimed at academic research, it is important to separate these from commercial grids. Scientific grids do not generally care much about protecting the data very well in the same ways that a company would want their data protected. The data used to perform scientific calculations is rarely of interest to anybody else than the scientists themselves. A current exception being the field of biology, in which there have been discussions regarding confidentiality and gridified calculation tasks. In the private

sector, on the other hand, where very large amounts of cash are spent each year on maintaining industrial secrets, the situation is quite different.

Companies using grid technologies usually want to keep the data secure and therefore rely on suppliers to deliver closed grid solutions. These closed solutions do not generally benefit the academic community. Open solutions provide the opportunity for anyone to add new features when needed, which is crucial in today's research. In this sense, the grids relevant to this thesis are considered to be scientific grids.

One of the most famous grids is SETI@Home. The software used in SETI@Home has evolved into the Berkley Open Infrastructure for Network Computing (BOINC) (University of California, 2009) and can be freely used to either support existing grids or create your own. BOINC utilizes the idle time of computers to perform calculations. In this grid form anybody can supply resources to the grid and get some form of credits in return. The actual amount of grid resource users is usually a very small minority compared to the amount of people adding resources to the BOINC grids.

Even if there have been some trials with Boinc-type grids, the average computing resources in a BOINC-type grid are not nearly powerful enough to support the data intensive calculations related to LHC experiments, since some calculations can require more than ten gigabytes of memory.

## 2.2 Structure of a Scientific Grid

"Grid is an infrastructure that involves the integrated and collaborative use of computers, networks, databases and scientific instruments owned and managed by multiple organizations." (Buyya & Srikumar, 2005)

### 2.2.1 Grid Components

A grid job is a term for a computational task, often consisting of a program or script, which is to be run on a grid resource. The resources in a scientific grid are usually computing elements (CE) or storage elements (SE). CEs allow users to run demanding software. SE's contain data that will be used by the tasks run on the computing elements. CE's are usually clusters and SE's are usually very large secondary memories. As seen in Figure 1, a cluster is an array of computers, or nodes, which are usually homogeneous except for the node commonly referred to as a frontend. The frontend acts as a kind of

gateway between the nodes and the rest of the world and is usually installed with software for managing the rest of the cluster. This allows the frontend to keep track of which nodes are available, down or busy with tasks. In order for a cluster to be used in a grid, essential grid middleware components have to be installed on the frontend.

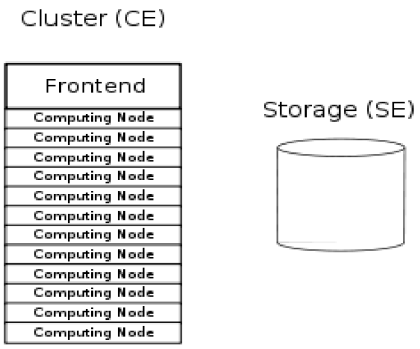



*Figure 1. A Computing Element (CE), consisting of a Cluster and its Frontend, and a Storage Element (SE). SEs are usually located near a CE so that the Internet connection does not become a bottleneck.*

In order to work efficiently, a grid has to keep track of the resources. This is usually done by implementing an information service, an entity that keeps track of the availability of the clusters, so that choices can be made regarding where to send grid jobs. The resource broker decides where grid jobs eventually will go. This broker fetches data from the information services and any possible authorisation databases, so that it can manage task queues to be as efficient as possible. The broker can be a separate entity or built into one of the other services or even into the grid client, from where the job originates. The authorisation databases keep track of use which virtual organisations are allowed to which resources and which user certificates belong to which virtual organisations. Figure 2 shows the flow of information from the CEs to the resource broker.

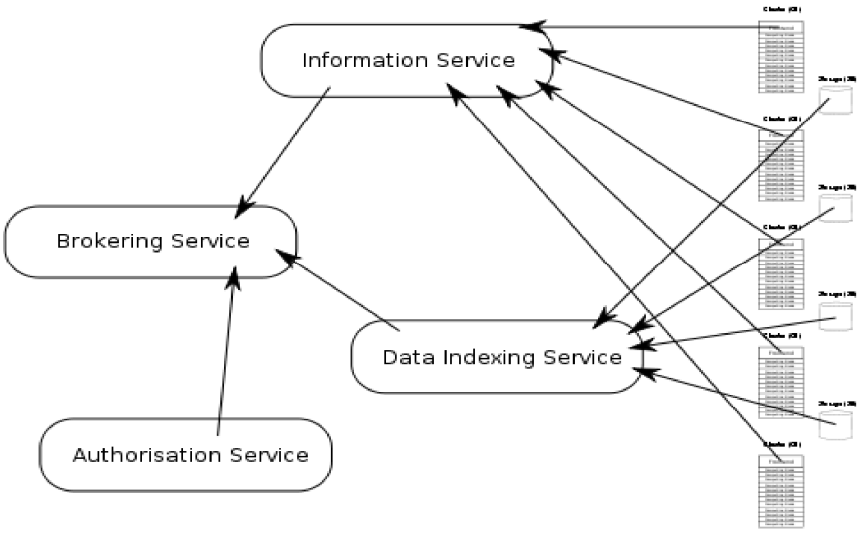



*Figure 2. The CEs send status updates to the Grid Information Service regarding activity on the CE. The SEs inform the Data Indexing Service about the location of different data sets. These services, along with the Authorisation Service, then let the Brokering Service know where a user can send a job and the queue data of the CE, respectively.*

### 2.2.2 Grid Layers

In order to gain a better understanding of a grid, we can divide it into four layers.

**The Network Layer**

At the very bottom of the grid, we have the network connections. The network layer covers all networking, from backbones to switchboards. Grids usually use the Internet to interconnect the different parts. This means that as this layer is improved in order to get more out of a grid, other Internet traffic can also benefit from it, and vice versa.

**The Resource Layer**

The resource layer consists of all pieces of equipment shared in some way amongst the users. This applies to computing resources, where mathematical tasks are sent to be run, as well as to sensors, from which the input data is collected, and to storage resources, where the data from the sensors can be stored for further analysis.

**The Middleware Layer**

The grid middleware is what ties this bundle together on a software level. This layer handles authentication, authorisation, queuing of tasks and monitoring of resource availability.

**The Application Layer**

The application layer is the only relevant layer for the grid user. The applications provide the user with the interface for sending of tasks, requesting data and presenting data.

## 2.3 Security Framework

### 2.3.1 Authentication

In order to allow for mutual authentication between the users and the services, x509 certificates signed by Certificate Authorities that govern the grid security are issued to the users. The certificates must be signed by a Certificate Authority (CA). This CA should be a CERN Trusted Certificate Authority. This will include the certificate in the CERN CA Certificate Chain which recognizes all parties involved in this particular grid.

A proxy certificate is a certificate that is signed either by a user certificate or by another proxy certificate. The idea behind these short-term certificates is to make the life of the grid user easier. Because users often send more than one job to the grid at a time, using proxy certificates saves the user from having to type the certificate pass phrase for each separate job. The subject of a proxy certificate looks identical to the original certificate with the exception of an added `/CN=proxy` to the end of the subject. (Globus, 2008)

Using proxy certificates, a user is able to assign jobs or Computing Elements to act on the user's behalf. This process is often referred to as Identity Delegation and is made possible by the fact that the proxy certificates have been signed by the user. That's why the proxy certificate is included when the job files are transferred to a computing element.

### 2.3.2 Authorisation

Virtual Organisations (VO) can be used to identify collaborations across traditional organisational borders and also access restriction to resources that are shared between different scientific organisations. In a grid environment, VOs can be used to associate both people and resources to a specific collaboration. The certificates are added to the relevant VOs so that the users gain access to the resources they need. It is possible to give single certificates permission to certain resources, but this is mainly used for testing purposes and not in a production environment.

Members from different organisations can join Virtual Organisations and be associated with them using the information available in their certificates. Restrictions can then be applied to the resources, allowing the members from certain VOs to access certain resources. Access to a resource can be allowed from one or several VOs. In order to manage all the VOs and permissions, the European DataGrid Project developed a system called Virtual Organization Membership Service (VOMS). (Globus, 2009) This system keeps track of the available VOs and their members.

## 2.4 Example of a job submission

The job is created by the user using a local application. A typical grid job would contain a configuration file which defines what command to run on the grid, what data to use if any, what kind of requirements there should be when picking a computing element (CE) on which to run the job, and possibly a raw whitelist and/or a blacklist restricting the choice of the CE. In addition to this configuration file the user can also include the file that is to be run by the command, defined in the configuration file, along with data needed to run this command. The data is usually transferred to the CE either directly from where the job is sent or then remotely from a storage element (SE).

The user then creates a grid proxy certificate using his or her own certificate. When the proxy certificate is created, the user can have it accredited by a VO by allowing a VOMS service add non-critical attributes to the proxy certificate before signing it. These attributes are what allow the user to use the resources assigned to the VOs that the user is associated with. After this, the proxy certificate is signed using the users own certificate. Signing the proxy certificate with the user certificate will require the passphrase of the user. (Sotomayor, 2004)

Once the job is ready to be sent, the middleware takes over. Before the job can be sent, a decision needs to be made regarding which CE the job will run on. This decision is usually referred to as job brokering and is depicted in Figure 3. Depending on the grid type and middleware being used, the job brokering is usually handled by the client or a separate brokering service. This brokering service checks with the Indexing Services which CEs are close to the SEs where the data needed for the job can be found and which queues contain how many jobs. The brokering service then compares this with the authorisation data fetched from an Authorisation Service in deciding upon a suitable CE.

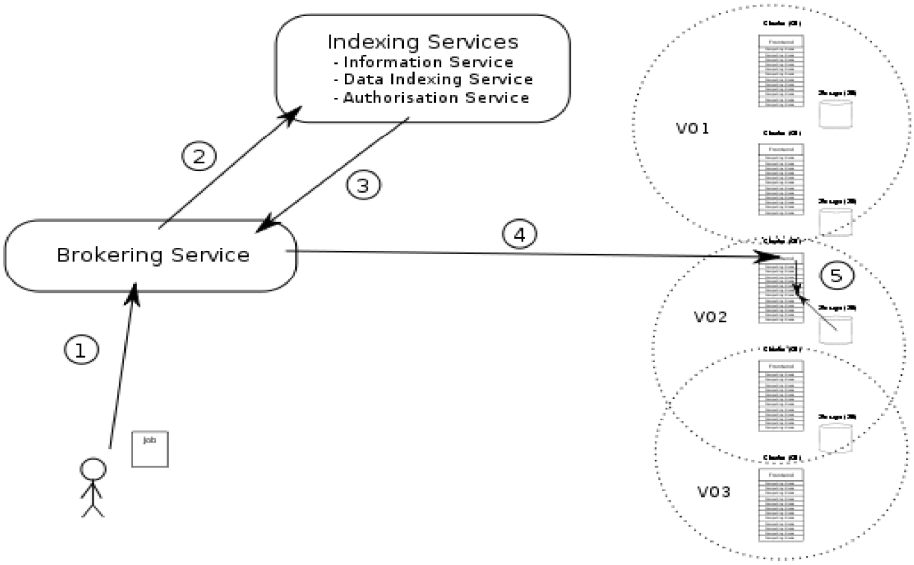



*Figure 3. This figure shows how a grid job finds its way to the resource on which it is to be run.*

When a CE has been decided on, the job is transferred to the chosen CE with a protocol chosen by the middleware. After this the frontend selects a node and sends the job to the selected node where the local batch system takes over. From this point on the middleware will keep track of the job and its status while it runs. The data needed for the job is also transferred to a temporary space on the node where the job is run. When the job has exited, the middleware registers this and makes the results ready for pickup.

The user can keep track of job progress through tools provided by the middleware. Statuses like ACCEPTED, FINISHED and FAILED will tell the user if the job has been accepted by a CE, if it has run and whether it has run successfully or failed. Once a job is complete, the user fetches the job from the grid using the middleware, either directly or through the local software.

The software used to create the job is usually found in the Application Layer. The VOMS service is a part of the Middleware Layer, as is the middleware itself and the software used to transfer the jobs or any necessary data. Even if the physical hardware, on which the job will actually run, exists in the Resource Layer, that resource will have a virtual manifestation in the middleware layer, namely the Computing Element. The Network Layer consists of all physical networking components, on which the traffic takes place.

# 3 ARC MIDDLEWARE

## 3.1 NorduGrid and ARC

"NorduGrid is a Grid Research and Development collaboration aiming at development, maintenance and support of the free Grid middleware, known as the Advances Resource Connector (ARC)." (NorduGrid, 2009)

The collaboration consists of scientific and academic organisations from several Nordic countries. NorduGrid has taken an active role in the High-performance Computing (HPC) community, communicating constantly with the user base of ARC. The collaboration aims to deliver a robust, portable and fully featured solution for a global Grid system and has taken a strong step towards these goals by developing and maintaining the ARC Grid Middleware.

The first ARC release was announced in May 2002. The grid middleware has since been deployed in several production environments. Emphasis is put on scalability, stability, reliability and performance of the middleware. A growing number of Grid projects, such as SweGrid, DCGC and NDGF, chose ARC as their middleware. One of the largest production Grids in the world is running on the ARC middleware. (NorduGrid, 2009)

At the HEPiX meeting in Umea in April 2009, Dr. Mathias Ellert presented NorduGrid's, then already active, project of making all NorduGrid's packages available in the repositories of the most popular Linux Distributions; Ubuntu, Debian and Fedora. This comes not only as welcomed news for the researchers, but also on some levels helps presenting grid technology and its uses to the Linux community. As of now, since NorduGrid is most active in the Nordic countries, this is where you find most of ARC's user base.

### 3.1.1 The use of ARC in Nordic Grids

M-Grid is the Finnish grid for Material Sciences. The grid was founded in 2004 as a joint project by the universities in Helsinki, Espoo, Turku, Tampere, Lappeenranta, Jyväskylä and Oulu. The idea of the grid was to combine the computing clusters of all of the universities into one single resource. The M-Grid resources are maintained by the organisations that own them, supported by the Finnish non-profit company CSC.

The M-Grid has been using the ARC middleware since the project began. Regardless of ARC's popularity in the Nordic countries, CERN has been using grid middleware implemented by gLite since 2006 when gLite 3.0 was released. These two grid middleware have few things in common and are therefore not very compatible. Since the resources in Finland use ARC and the default grid middleware in CERN is incompatible, modifications have to be added to most grid related software if the Finnish resources are to be used.

### 3.1.2 ARC Components

According to the NorduGrid website the three main components of ARC are the Grid Services, The Indexing Services for data and resources, and the ARC Client capable of making intelligent use of the distributed information available on the grid. The Grid Services consist of three parts: the Grid Manager, gridftpd and the Information Services. The Grid Manager is the service running on the computing element which handles the jobs and processes input and output files. Gridftpd is a service for handling data transfers over the gridftp protocol. The Information Services are entities that keep track of the status of the computing elements, the storage elements as well as the authorisation data.

## 3.2 Using ARC

The main part of an ARC job is the configuration file, which is written in xRSL format. xRSL stands for Extended Resource Specification Language. RSL is a reference to the configuration language used in the Globus Toolkit, which is a set of tools that ARC used to rely heavily on. As for most grid solutions, the configuration file defines the job parameters regarding what command to run, which command parameters to use, preferences regarding where to run it, and also what data to run it on. When the xRSL is complete, the job can be submitted using the command supplied by the ARC client, namely ngsub. This command starts the procedure of picking a CE before sending the job files there to be run and then returns a jobID to the user.

The CMS data usually already exists on Storage Elements (SE) around the world, stored in sets of data. The xRSL definition regarding what data to use can be set as a dataset path or as a gridftp URL. Usually the dataset paths are used and the ARC client will use the Grid Index Information System (GIIS) to figure out which CEs lie close to SEs where

the data can be found. The vicinity of the two elements is defined by the network topology and therefore also by the geographical locations of the two elements. In order to get the most out of the hardware used, the goal is usually to run the job on a CE where the data sets can be found within the same building, hopefully even in the same local network.

As soon as a job is fully transferred to a CE, ARC starts monitoring the state of the job. If there is data that the job needs, then ARC transfers this data to the temporary space where the job is to be run. When the job has stopped running, either due to success or failure, ARC marks the job as stopped and sends the user an email if an email address was supplied in the xRSL file. The user can then use the ARC client command ngget, along with the jobID, to have ARC fetch the result and clean up after the job by deleting files on the CE that are no longer needed.

The differences between ARC and gLite can already be noticed in the first step. While ARC uses xRSL, which is written in a more traditional configuration format without any complex structure, gLite uses JDL, which is written in a language similar to XML. Even if the user does not notice much difference after this, the grid job scheduling with gLite is not done in the client as with ARC. gLite uses something called a Workload Management System (WMS) for deciding which CE to send the job to. While ARC uses GIIS as an information system keeping track of the CEs, SEs and their attributes, gLite has an indexing system of its own called Berkley Database Information Index (BDII), which it uses to store the attributes of the resources. BDII and GIIS both use LDAP for handling the information but their LDAP schemes are different.

# 4 CMS ANALYSIS SOFTWARE

"CMS presents challenges not only in terms of the physics to discover and the detector to build and operate, but also in terms of the data volume and the necessary computing resources. Data sets and resource requirements are at least an order of magnitude larger than in previous experiments." (Lassila-Perini, CMS Computing Model, 2009)
Since the amount of data handled in CMS experiments is so vast, the infrastructure behind the computing is very intricate.

## 4.1 Data type tiers

The storing of the CMS data sets is divided into tiers, but these are not to be confused with the data tiers that represent the levels of data that is stored. The three main data tiers are as follows:

1. RAW – Full event information from the Tier-0 storage, containing "raw" data.
2. RECO – Reconstructed data generated by a first-pass processing of the raw data. RECO data can be used for analysis, but is too large for frequent or heavy use.
3. AOD – Analysis Object Data is a "distilled" version of the RECO event information and is expected to be used for most analyses. AOD provides a trade-off between event size and complexity of the available information to optimize flexibility and speed for analyses.

(Lassila-Perini, CMS Computing Model, 2009)

## 4.2 CMSSW - CMS Software

The CMS Software (CMSSW) is the software used to analyse the data sets. The input files for CMSSW include CMS data sets, mainly in the RECO and AOD forms, and a configuration file defining which files to use. When CMSSW has finished the analysis, the output can be found in numerous smaller files. These can be analysed either with command-line interface (CLI) tools in a Unix shell or graphically presented with software like edmBrowser or Fireworks.

The KHC started producing collision data in November 2009. Years before the LHC started producing actual data, test sets of data were being produced through a Monte Carlo (MC) procedure. These data sets are used on the collision data for research.
CMSSW is still being developed further and new versions are published all the time. This is an important factor for the end-user, because the MC data set that are generated only work for certain versions of CMSSW. This mainly means that old MC data does not work with new versions of CMSSW and vice versa. The online guide to CMSSW says that this limitation will subside as the code for CMSSW stabilises more. (Lassila-Perini, CMS Computing Model, 2009)

## 4.3 CRAB - CMS Remote Analysis Builder

The main task of the CMS Remote Analysis Builder (CRAB) is to serve as a frontend, or user interface, for the grid. CRAB handles the packaging of the analysis job and then hands the job over to CMSSW, either directly or through grid middleware. After the job has been submitted, CRAB can be used for checking the status of the job and fetching the results when the job is executed.
Like many applications used in a grid environment, CRAB consists of a number of smaller parts. The grid job transmitting part of CRAB originally belonged to a project called ProdAgent. (Evans, 2008) ProdCommon has become a self-sufficient library for several grid applications including the Python files that create the scheduler object. This object handles the communication between CRAB and a specific grid middleware.

### 4.3.1 The Structure of CRAB

CRAB is executed entirely from a shell script called crab, which basically is a wrapper for the Python file crab.py. This Python script checks the parameters and calls the necessary subroutines accordingly. When run, crab.py will create an object called crab, resolve which action parameter was used and run its class method called run(). This method will then import the Python file associated with the action and run a method from that file.
CRAB comes bundled with a few different schedulers, each written for a separate middleware configuration. The scheduler type can be chosen globally, by configuring the main crab.cfg in the installation directory, or it can be set separately for a stack of jobs.

Normally each set of jobs has its own configuration file containing information about which CMS datasets to use. This same file can contain a scheduler directive, telling CRAB which scheduler to use for this particular batch of jobs.

Aside from this native code, CRAB consists of a significant amount of other software packages, for example ProdCommon, SQLite and OpenSSL. As you can see in Figure 4, the external part of CRAB is much larger than the folder named `python/`, which contains the native code.

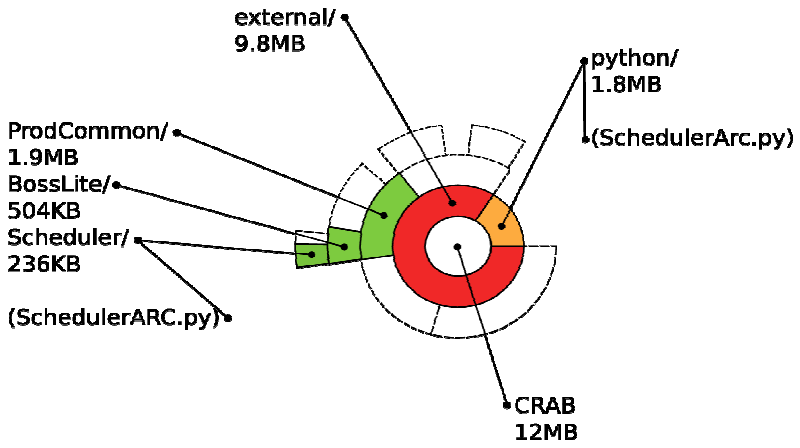



*Figure 4. The file structure of CRAB. The circles represent folder depth and the slicing signifies separate folders. The Solid lines represent the folders that are relevant to this thesis.*

### 4.3.2 Inner workings of CRAB

The rest of this chapter aims at giving a deeper glance into how CRAB works. Unless specifically stated as a name of a file, during the rest of this chapter "crab" refers to the Python object with this name and CRAB will refer to the software as a whole.

`crab.py` is run with command line parameters. `crab.py` creates a crab object and executes crab.run() which then runs initialize_() which in turn runs initializeActions_() to check for action parameters like -create or -submit. Each action parameter creates an object with the parameter's name using the appropriate Python file. The objects are then added to the crab.actions array. Because of the various names, the object is referred to as

ActionObject in Figure 5. Examples of some of these Python files would be Submitter.py, SubmitterServer.py, GetOutput.py, GetOutputServer.py, Status.py, Killer.py, Creator.py and Cleaner.py. In Figure 5 these files are named "<action>er.py". The names of these files are mostly self-explanatory and the list of available action parameters can be found in the user manual. The Server suffix is used for some files which will be imported instead of the normal one in such a case that the CRAB installation has been configured to use a CRAB server. Such implementations are however outside the scope of this thesis.

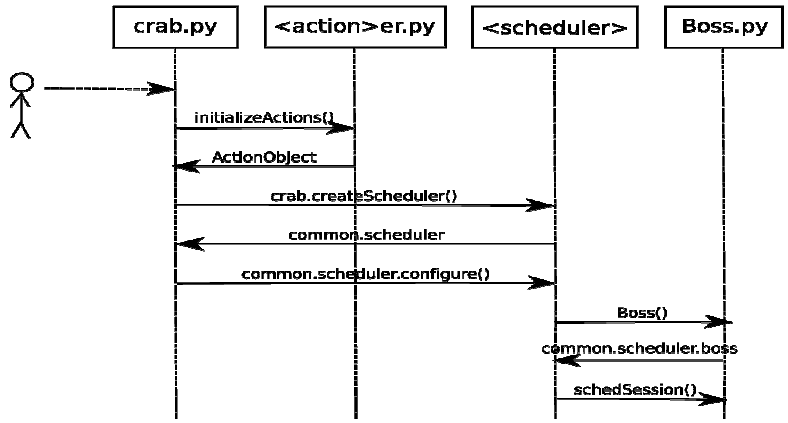



*Figure 5. The order in which CRAB methods are called from the different files within the* `python/` *folder.*

Once the actions have been determined and their respective objects created, crab creates an object for a scheduler using the method crab.createScheduler(). This method reads the name of the scheduler from the configuration variables, tries to import that scheduler, and defines common.scheduler to be an object of that type. When done, common.scheduler.configure() is run from the scheduler in the `python/` -folder, for example from SchedulerArc.py. In Figure 5 the name of this file is represented by the text <scheduler>.

Depending on which scheduler is used, the newly created scheduler will inherit methods and variables from various other schedulers. The current set of schedulers in the CRAB package makes up a tree-like inheritance structure. All schedulers also inherit `python/Scheduler.py` at the end of the inheritance lineage. The `Scheduler.py` constructor will define a variable called common.scheduler._boss to be an instance of the

Boss class. This is one of the key places where CRAB has to be altered in order to allow the use of new schedulers. The constructor for the Boss class defines a map named SchedMap and uses this to figure out which file to import from the folder `external/ProdCommon/BossLite/Schedulers`.

The Boss object, defined as common.scheduler._boss, will include wrapping methods for the different CRAB actions like submit(), getOutput() and cancel(). These action methods will handle debug logging and other BossLite-related tasks and then refer to a method called common.scheduler._boss.schedSession().

The session method will initialize and return a session object defined in the file `external/ProdCommon/BossLite/Scheduler/Scheduler.py` and send the filename of the configured schedule, which was derived using SchedMap, along as a parameter. This temporary Boss.session object is defined in BossLiteAPISched.py in the API part of the BossLite package. BossLiteAPISched.scheduler is defined as the Scheduler.py which is included in the ProdCommon package. This Scheduler defines an object called schedObj which will be defined in the file mentioned in the constructor parameter, for example SchedulerARC(.py). In order to understand this better, one could refer to SchedObj as common.scheduler._boss.session.scheduler.SchedObj, but because of how the _boss.session object is created only when needed, this is never done this way.

When the initialization routines have been run, crab.run() is executed. This method loops through the array of command line action parameters, executing the run() method for every action parameter for each object created earlier. Doing this enables the combining of several action parameters during one single CRAB execution; e.g. creating and submitting a job with a single CRAB execution. These action-specific run() methods then refer to the common.scheduler methods if and when they need access to the scheduler functions. For example, when Submitter.py submits a job it uses common.scheduler.submit() with the job as a parameter.

# 5 CRAB SCHEDULERS

While CRAB is made up of a significant amount of smaller packages, most of which could be called "plug-ins", one specific group of these exclusively handles the communication with the software managing the computing resources. Usually this software is a grid middleware. CRAB refers to these plugins as schedulers. A scheduler consists of, at least, two different files, one located in the ProdCommon package in the BossLite section, and another located in the CRAB package in the python/ folder. ProdCommon is a package of libraries used for communicating with a grid middleware. It is actually a project of its own and is therefore found separately in the CERN CVS, next to CRAB. SchedulerARC.py, found in the ProdCommon package, contains low level methods for direct interaction with the grid middleware. SchedulerArc.py, on the other hand, which can be found in the python/ folder, contains high level methods, which handle configuration data and prepare it for the scheduler object. Figure 6 shows the how the different objects are created in relation to each other.

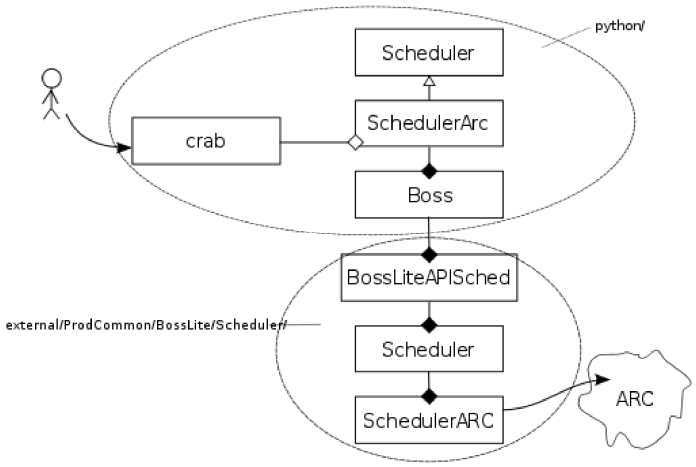



*Figure 6. The relation between two different scheduler files and the folders in which they reside.*

## 5.1 SchedulerFake

No actual guide or tutorial exists on how to write a CRAB scheduler plugin. The documentation on how to use the existing version of CRAB in the CERN TWiki (Fanzago, 2007) is regularly updated along with the software and the development team is active in replying to questions about the software. This development team has also been kind enough to add a dummy scheduler to the package, called SchedulerFake. While SchedulerFake only supplies the guidance to roughly half of the complete plugin, it still sheds some light on the methods that are expected to be found in the scheduler file in BossLite.

Creating a SchedulerARC.py file using SchedulerFake is a natural first step. This gives a general direction on what structure to use and on which methods are required. Following the structure in SchedulerFake, each new scheduler is defined within its own class. This class will be the main class for the scheduler; it will contain most of the necessary methods and definitions, and is to be named after the file. In this case the scheduler was named SchedulerARC.

## 5.2 CRABARC

### 5.2.1 SchedulerARC.py

SchedulerARC.py contains the low-level methods that directly interact with the grid middleware. The methods required in any ProdCommon Scheduler file are listed in SchedulerFake and are as follows:

**def __init__( self, **args )**

This method initializes the object and defines the necessary variables and their values.

**def submit( self, task, requirements=”, config =”, service=” )**

This method is run when CRAB wants to start sending the job towards the computing element. This method is the only required submission method for a Scheduler. However, quite a few schedulers use this method mainly for preparing the job. SchedulerARC uses submitJob() for communicating with the grid middleware and for performing the actual sending.

**def getOutput( self, obj, outdir=” )**

This method is run when a user wants to fetch the output of an analysis. It iterates through all the jobs and runs getJobOutput() for each valid job. getJobOutput() then performs the actual fetching from the grid.

**def kill( self, obj )**

This method runs when a user requests a job to be aborted.

**def purgeService( self, obj )**

This method clears the user space on the resources where a certain job has been sent.

**def matchResources( self, obj, requirements=”, config=”, service=” )**

This method looks for suitable resources to be used for an analysis. It should also take authorisation into account and check if the user is allowed to use the resources that are listed as available.

**def postMortem( self, schedulerId, outfile, service)**

This method fetches and saves any additional logging that occurs after a job has died, regardless of how it died.

**def query(self, obj, service=”, objType=’node’)**

This method is for querying the status of the scheduler and of any possible active jobs.

**def jobDescription ( self, obj, requirements=”, config=”, service = ” )**

This method returns a scheduler-specific job description.

**def decode ( self, task, requirements=” )**

This method prepares the configuration parameters for a job. In the case of CRABARC it returns the options in an ARC friendly xRSL format. The xRSL data defines which CMS datasets to use for the analysis. The xRSL data may contain restrictions regarding the choice of resources in the form of a whitelist, a blacklist or both.

Aside from these required methods already now written in SchedulerARC.py, there are some additional functions used within the scheduler object for better structure.

**def submitJob(self, task, job, requirements)**

This method extracts from the job configuration options, which are relevant to the submitting procedure, and adds these option parameters to the command that submits the job. This method is run from submit().

**def getJobOutput(self, job, outdir)**

This method downloads a job from the grid and cleans up after it. This method is run from getOutput().

**def query_giis(self, giis)**

This method uses the GIIS server given as a parameter and fetches information about the CEs and any possible sub-GIIS -servers. This method is used for fetching all possible CEs and building a tree of GIIS servers.

**def check_CEs(self, CEs, tags, vos, seList, blacklist, whitelist, full)**

This method goes through a list of CEs, compares it to the parameters given and returns a list of valid CEs.

**def pick_CEs_from_giis_trees(self, root, tags, vos, seList, blacklist, whitelist, full)**

This method uses query_giis() to build up a tree of services, runs check_CEs() to match for suitable CEs and builds a full list of accepted CEs using the full GIIS tree before returning the list to lcgInfo().

**def lcgInfo(self, tags, vos, seList=None, blacklist=None, whitelist=None, full=False)**

This method is the method run from Boss.py and fetches CE information. This starts the procedure of creating a list of suitable CEs. Since CRAB also does some brokering of its own, it is important that the attribute `nordugrid-cluster-localse` of the CE is set as the SE which will contain the data that is to be analyzed.

### 5.2.2 SchedulerArc.py

While SchedulerARC.py contains the major part of the scheduler, it is roughly only 75% of the scheduler. In the CRAB package in the python/ folder there will also be a similarly named file, namely SchedulerArc.py. Where SchedulerARC.py interacts with the middleware, SchedulerArc.py works with CRAB, preparing the job and the scheduler using the configuration values.

**def __init__(self, name='ARC')**

This method, the constructor, actually does nothing more in SchedulerArc.py than name the scheduler.

**def envUniqueID(self)**

This method creates an ID for a job, making the tracking of it a lot easier. SchedulerArc.py uses arcId for this.

**def realSchedParams(self,cfg_params)**

This method returns a set of valid configuration parameters.

**def configure(self,cfg_params)**

This method is run in crab.py just after the scheduler has been created and contains workarounds for certain issues regarding global or environment variables. (Namely EDG_WL_LOCATION and X509_USER_PROXY.)

**def checkProxy(self, minTime=10)**

This method checks if there is a valid grid proxy running and creates one if there isn't.

**def ce_list(self)**

This method creates the whitelist and blacklist for Computing Elements using the BlackWhiteListParser in the WMCore package located in the external/ folder of CRAB.

**def se_list(self, id, dest)**

This method creates the whitelist and blacklist for Storage Elements using the BlackWhiteListParser in the WMCore package located in the external/ folder of CRAB.

**def sched_parameter(self,i,task)**

This method returns a list of required runtime configuration parameters. In SchedulerArc.py, this is done by letting this method act as a wrapper for the methods runtimeXrsl() and clusterXrsl(), concatenating their return values.

**def runtimeXrsl(self,i,task)**

This method returns an Xrsl code snippet with the required runtime variables.

**def clusterXrsl(self,i,task)**

This method returns an Xrsl code snippet which defines which SE and CE to use.

**def wsInitialEnvironment(self)**

This method sets the commands that are to be run before a job is set to run at the CE.

**def wsExitFunc(self)**

This method sets the commands that are to be run after a job has run at a CE.

**def tags(self)**

This method tags the job according to its task. Certain types of jobs have certain priority settings on some clusters.

**def loggingInfo(self,list_id,outfile )**

This method returns logging info about a job.

### 5.2.3 CRAB Action Parameters

As mentioned before, CRAB is executed from a single executable. The actions of this file named crab.py are defined through the action parameters. Since the list of available

parameters can be found in the manual, they can easily be used for creating a crude test to see if the plugin works as intended.

The most relevant parameters are:

**create**

This parameter creates the job using the configuration parameters in crab.cfg.

**submit**

This parameter sends the job to the computing element.

**status**

This parameter checks the current status of all jobs.

**getoutput**

This parameter fetches the output from the finished jobs. If there are jobs still running that haven't been set as finished, their output isn't fetched.

**kill**

This parameter aborts any unfinished jobs. If the jobs are running they will be killed.

**clean**

This parameter cleans up any traces of a job that has been run. It is usually run if some jobs have failed.

**cfg fname**

This parameter allows the user to use an alternative name for the configuration file.

## 5.3 Using a Scheduler to Sumbit a Job

There are several steps in submitting a job to the grid using CRAB. Throughout these examples CRABARC will refer to the scheduler as a whole and CRAB will refer to the whole software. SchedulerArc.py and SchedulerARC.py are not the same files and exist in separate folders, as is previously mentioned in this chapter.

### 5.3.1 Creating a job

In order to create a CRAB job at least one set of data has to be chosen and configured in the job specific crab.cfg. When the configuration file has been written and all relevant information defined, the user runs crab with the action parameter '-create'. After the action parameter has been identified, CRAB will create a Creator object that uses the common libraries to bundle up the files needed to run the job, and add these files to a

database associated with the job as well as create the script to be run once the job has been submitted.

As a final step of the initialisation, CRAB will create an object for a scheduler. The method createScheduler() will read the given configuration options and attempt to import a file based on the string given as the scheduler name in the configuration. In the case of CRABARC, this means that common.scheduler will be set as the object defined in the file python/SchedulerArc.py. This object, just like its Condor and gLite counterparts, extends the SchedulerGrid object which in turn extends the Scheduler object. Because of this, all schedulers define a common.scheduler._boss object as Boss, which is defined in the `/python` folder.

When CRAB performs the run() method of the actions defined by the command line parameters, the Creator object will perform its main function preparing the job. The only thing in the Creator object that is relevant to the schedulers, is when it calls the common.scheduler.sched_fix_parameter() and common.scheduler.declare() methods. The sched_fix_parameter() is defined in SchedulerGrid.py and will add scheduler-specific parameters to the job information in the database. As for declare(), it is not specifically defined in SchedulerArc.py and will fall back to Scheduler.py, where it is defined as a wrapper for common.scheduler._boss.declare(). Boss.declare() will add the list of jobs and some job parameters to the database. There is nothing else happening in the Creator object that is relevant to the scheduler.

### 5.3.2 Submitting a job

Initialisation tasks will be the same for this step as for the job creation step.

When Submitter.run() is run, it will build up a list of jobs to send based on the contents of the database and then run Submitter.performSubmission(). This method will try to run delegateProxy() which is only defined in the gLite scheduler and therefore not relevant to this CRABARC example. After this, performSubmission() will loop through the array of jobs and submit them all, one at a time, using common.scheduler.submit().

The submit() method is defined in Scheduler.py in `python/`, where it again performs a mere logging action before running the boss.submit() method. The Boss object will refer to the Boss.session.submit() method defined in BossLiteAPISched.py, which lies in `external/ProdCommon/BossLite/API/`, making the transition from the native code in `python/` to the ProdCommon package in `external/`.

The BossLiteAPISched.submit() method will call upon the BossLiteAPISched.scheduler object in `external/ProdCommon/BossLite/Scheduler/Scheduler.py`, which will then use the schedObj and call its submit() method. Since schedObj is defined as the configured scheduler, it will now use SchedulerARC.py.

The submit() method in SchedulerARC.py, splits up the jobs into smaller parts and uses submitJob() to submit them, one part at a time. This method will fetch job configuration options and use get_ngsub_opts() to format these options so that they can be appended as command line arguments when the job is submitted using the ARC client's `ngsub` command.

### 5.3.3 Fetching the results

Just as with job creation and job submission, fetching the results includes the same initialisation procedures. The action object for fetching results is defined in GetOutput.py. GetOutput.GetOutput(), called from its run() method, checks for available space first, unless the user has specifically configured it not to check for it, and raises an exception if it deems the available space to be insufficient. After checking this, common.scheduler.getOutput() is called.

As with the submission example, the method wrapping chain is as follows: common.scheduler.getOutput(), Boss.getOutput(), BossLiteAPISched.getOutput(), Scheduler.getOutput() and schedObj.getOutput(). The last of these is defined in SchedulerARC.py which uses getJobOutput() for the actual fetching with the ARC client commands. The normal way to fetch the results with the ARC client would be by using `ngget`, but this places the fetched results in a directory structure decided upon by the ARC client instead of using the same structure which the other schedulers use. Instead, getJobOutput() uses `ngcp` with the destination directory as a parameter to copy the results from the grid and then runs `ngclean` to erase the results from the grid resource.

# 6 CONCLUSIONS

At this moment the scheduler plugin CRABARC is working and is in production use. The plugin has been in production use now for a few months and has successfully sent more than 7000 jobs across the grid. There are fewer and fewer bug reports sent in each month and the few bug reports that are filed are handled fairly quickly. Since CRABARC is now officially a part of the CRAB package, and is being developed in the CRAB part of the CERN CVS, the project can be deemed successful.

Even if CRABARC is now considered a fully working piece of software, there are still aspects that need improving. At the time of writing, the plugin still uses the command line ARC client commands. So in order for the plugin to work, the client environment needs to have the ARC client installed. Eventually the plan is to overcome this by implementing ARCLIB, a Python library of ARC functions. At this moment this is not possible to do because of Python version incompatibility issues. ARCLIB uses the system installation of Python while CRAB sticks to the version installed with CMSSW. No schedule has been set for the implementation of this, but the development on this is expected to be easier once the development of CMSSW stabilises.

During the meetings regarding the ARC plugin, there has been talk about making a larger scheduler which would enable communicating with a number of different grid middleware implementations before deciding which grid middleware to use. This new Scheduler would supersede most of the other Schedulers, possibly all of them. The discussions have been about either making a new separate Scheduler, based on CRABARC, or alternatively turning CRABARC into this universal Scheduler.

Having participated in a project such as this has given great insight into how the grid works and how software can be written to use plugins.

## REFERENCES


Buyya, R., & Srikumar, V. (2005). A Gentle Introduction to Grid Computing and Technologies.

CERN. (2009, 07 20). *CERN - European Organization for Nuclear Research*. Retrieved 07 30, 2009, from CERN - European Organization for Nuclear Research: http://public.web.cern.ch/public/

CERN. (2009, 12 1). *Compact Muon Solenoid*. Retrieved 2 18, 2010, from CERN: http://cms.web.cern.ch/

CERN. (2009, 07 30). *DBS Data Discovery Page*. Retrieved 07 30, 2009, from DBS Data Discovery Page: https://cmsweb.cern.ch/dbs_discovery/

CERN. (2009, 11 27). *LHC_Homepage*. Retrieved 2 18, 2010, from CERN: http://lhc.web.cern.ch/

Debian GNU/Linux. (2009, 07 29). *Debian - The Official Webpage*. Retrieved 07 30, 2009, from Debian - The Official Webpage: http://debian.org/

EGEE. (2004, 12 31). *European Grid Computing Changes Gear*. Retrieved 10 07, 2009, from European Grid Computing Changes Gear: http://www.dante.net/server/show/conWebDoc.1255

EGEE. (2008, 01 01). *gLite*. Retrieved 11 20, 2009, from gLite: http://glite.web.cern.ch/glite/

Evans, D. (2008, 1 5). *ProdCommon*. Retrieved 10 20, 2009, from CERN TWiki: https://twiki.cern.ch/twiki/bin/view/CMS/ProdCommon

Fanzago, F. (2007, 7 20). *CRAB How-to*. Retrieved 10 20, 2009, from CERN TWiki: https://twiki.cern.ch/twiki/bin/viewauth/CMS/WorkBookRunningGrid

Fermilab. (2009, 05 01). *Scientific Linux - The Official Webpage*. Retrieved 07 30, 2009, from Scientific Linux - The Official Webpage: https://www.scientificlinux.org/

Foster, I. (2002). What is the Grid? A Three Point Checklist. 4.

Globus. (2009, 10 07). *European Data Grid*. Retrieved 10 07, 2009, from European Data Grid: http://www.globus.org/alliance/news/EDG-index.html

Globus. (2008, 01 01). *Signing Onto the Grid: Creating a Proxy Certificate*. Retrieved 1 29, 2009, from The Globus Alliance: http://www.globus.org/security/proxy.html

Globus. (2009, 10 07). *VOMS*. Retrieved 10 07, 2009, from European Data Grid: http://www.globus.org/grid_software/security/voms.php

Grey, F. (2009, 07 30). *Grid Café*. Retrieved 07 30, 2009, from Grid Café:

http://www.gridcafe.org/

Lassila-Perini, K. (2009, 5 15). *CMS Computing Model*. Retrieved 10 16, 2009, from CERN TWiki: https://twiki.cern.ch/twiki/bin/view/CMS/WorkBookWhichRelease

Lassila-Perini, K. (2009, 12 3). *CMS Computing Model*. Retrieved 10 16, 2009, from CERN TWiki: https://twiki.cern.ch/twiki/bin/view/CMS/WorkBookComputingModel

NorduGrid. (2009, 06 08). *NorduGrid*. Retrieved 07 30, 2009, from NorduGrid: http://www.nordugrid.org/

Professor Malcolm Atkinson, e.-S. E. (2009, 12 07). *e-Science*. Retrieved 12 07, 2009, from e-Science: http://www.rcuk.ac.uk/escience/default.htm

Redhat. (2009, 07 30). *Redhat Enterprise Linux - The Official Webpage*. Retrieved 07 30, 2009, from Redhat Enterprise Linux - The Official Webpage: http://www.redhat.com/rhel/

Sotomayor, B. (2004, 1 1). *Delegation and single sign-on (proxy certificates)*. Retrieved 2 4, 2010, from The Globus Toolkit 4 Programmer's Tutorial: http://gdp.globus.org/gt4-tutorial/multiplehtml/ch10s05.html

Spiga, D. (2009, 07 24). *The CMS Wiki: SWGuideCrab*. Retrieved 07 30, 2009, from CERN TWiki: https://twiki.cern.ch/twiki/bin/view/CMS/SWGuideCrab

University of California. (2009, 09 04). *BOINC*. Retrieved 09 28, 2009, from BOINC: http://boinc.berkeley.edu/

## APPENDIX 1. EXAMPLE XRSL FILE

```
&
(* some local variables defined for further convenience *)
     (rsl_substitution=("TOPDIR" "/home/johndoe"))
     (rsl_substitution=("NGTEST" $(TOPDIR)/ngtest))
     (rsl_substitution=("BIGFILE"
"/scratch/johndoe/100mb.tmp"))
(* some environment variables, to be used by the job *)
     (environment=("CMS" "/opt/CMSSW") ("CERN" "/cern"))
(* the main executable file to be staged in and submitted to
the PBS *)
     (executable="checkall.sh")
(* the arguments for the executable above *)
     (arguments="pal")
(* files to be staged in before the execution *)
     (inputFiles = ("be_kaons" "")
     ("file1" gsiftp://grid.uio.no$(TOPDIR)/remfile.txt)
     ("bigfile.dat" $(BIGFILE) )    )
(* files to be given executable permissions after staging in
*)
     (executables="be_kaons")
(* files to be staged out after the execution *)
     (outputFiles=
        ("file1" "gsiftp://grid.tsl.uu.se/tmp/file1.tmp")
        ("100mb.tmp"
"rls://rls.nordugrid.org:39281/test/bigfile")
        ("be_kaons.hbook"
gsiftp://ce1.grid.org$(NGTEST)/kaons.hbook)   )
(* user-specified job name *)
     (jobName="NGtest")
(* standard input file *)
     (stdin="myinput.dat")
(* standard output file *)
```

```
     (stdout="myoutput.dat")
(* standard error file *)
     (stderr="myerror.dat")
(* GM logs directory name *)
     (gmlog="gmlog")
(* flag whether to merge stdout and stderr *)
     (join="no")
(* request e-mail notification on status change *)
     (notify="bqfe jesper.koivumaki@host.fi")
(* maximal CPU time required for the job, minutes for PBS*)
     (CpuTime="60")
(* maximal time for the session directory to exist on the
remote node, days *)
     (lifeTime="7")
 (* memory required for the job, Mbytes *)
     (Memory="200")
(* wall time to start job processing *)
     (startTime="2002-04-28 17:15:00")
(* disk space required for the job, Mbytes *)
     (Disk="500")
(* required architecture of the execution node *)
     (architecture="i686")
(* required run-time environment *)
     (runTimeEnvironment="APPS/HEP/Atlas-1.1")
(* number of re-runs, in case of a system failure *)
     (rerun="2")
```

## APPENDIX 2. EXAMPLE CRAB.CFG FILE

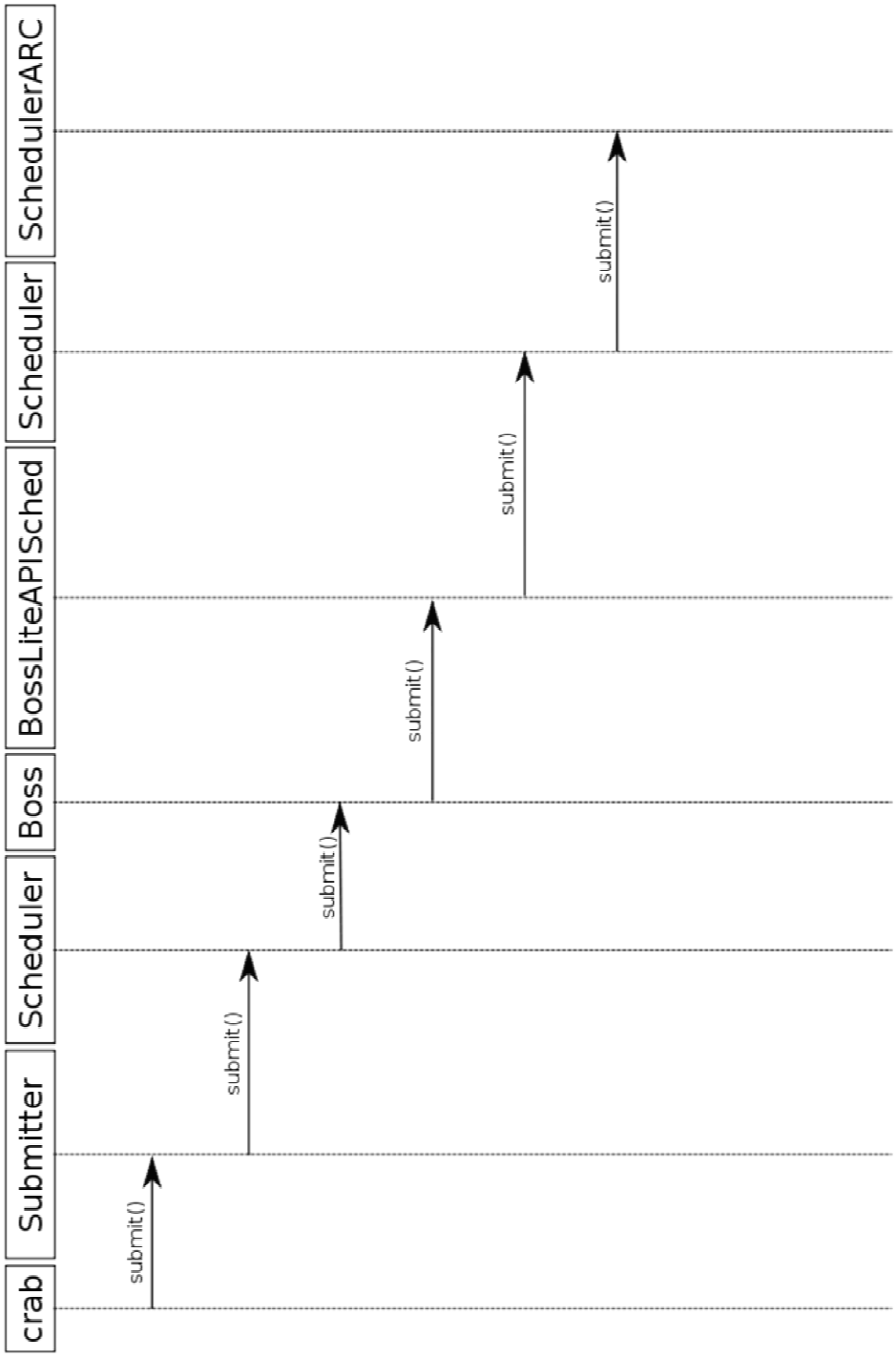
```
[CMSSW]
total_number_of_events=1000
number_of_jobs=3
pset=patLayer1_fromAOD_full.cfg.py
#datasetpath=/RelValZEE/CMSSW_2_1_9_STARTUP_V7_v2/GEN-SIM-
DIGI-RAW-HLTDEBUG-RECO
datasetpath=/QCDDiJetPt380to470/Summer08_IDEAL_V9_v1/GEN-
SIM-RECO
output_file=PATLayer1_Output.fromAOD_full.root

[USER]
return_data=1
email=jesper.koivumaki@host.fi

[CRAB]
#scheduler=glite
scheduler=arc
jobtype=cmssw
#server_name = bari
server_name = None

[EDG]
SE_white_list = madhatter.csc.fi
CE_white_list = sepeli.csc.fi,ametisti.grid.helsinki.fi
```

## APPENDIX 3. CRAB SCHEDULER FLOW DIAGRAM




**7**