PROCEEDINGS OF SCIENCE

Powering Distributed Applications with DIRAC Engine

Víctor Méndez Muñoz
Computer Architecture and Operating System Department, Universitat Autònoma de Barcelona
Campus UAB, Edifici Q, 08193 Bellaterra (Barcelona), Spain.
E-mail: vmendez@caos.uab.es

Adria Casajus Ramo and Ricardo Graciani Diaz
Department of Structure and Constituents of Matter, University of Barcelona
Diagonal 647, ES-08028 Barcelona, Spain
E-mail: graciani@ecm.ub.edu and adria@ecm.ub.edu

Andrei Tsaregorodtsev
Centre de Physique des Particules de Marseille, CPPM - CNRS
163 Av de Luminy Case 902 13288 Marseille, France
E-mail: atsareg@in2p3.fr

DIRAC has evolved from a framework for distributed computing matters operated by the final user, to a platform that can also be used from third-party Virtual Research Environments (VRE), Science Gateways (SG) or distributed computing applications. Therefore, DIRAC has become a platform providing distributed computing interoperability and high level services for science. For this purpose, DIRAC supports several Application Programming Interfaces (API) designed for a versatile programming model in different developing scenarios. This paper presents such scenarios categorized by the location of the API client. Thus, a first scenario is an application server powered by DIRAC services. Second, a web browser running embedded DIRAC calls, for dynamic web client composition. Third scenario is a DIRAC Platform as a Service (PaaS) powering distributed applications with High Throughput Computing (HTC) on the client side. This paper describes the underlying DIRAC architecture and the main functionalities of each API and developing scenario. Overall DIRAC is a coherent toolkit that enables a wide range of component developments like service for scientific applications, web plugs or PaaS agents.

The International Symposium on Grids and Clouds (ISGC) 2014
March 23-28, 2014
Academia Sinica, Taipei, Taiwan

PoS(ISGC2014)042

1. Introduction

DIRAC is a software project [1] that provides a general-purpose framework for building distributed computing systems. It is used now in several high-energy physics and astrophysics experiments as well as by user communities in other scientific domains. After more than 10 years in production, DIRAC software has evolved considerably from its original form, complemented by extension modules providing Web and REST interfaces, connection to Clouds, Documentation, and others with specific functionalities for particular communities (LHCb, ILC, CTA, Belle II, BES III, etc.). Furthermore, several DIRAC instances are running multi-community portals addressing medium-size communities and the long tail of science. This allows economies of scale in software management and operational effort (DIRAC 4 EGI pilot portal @ EGI (E.U.), DIRAC 4 FranceGrilles @ IN2P3 (France), DIRAC 4 IHEP @ IHEP (China), and others).

For many years now, DIRAC includes the necessary components to build a complete distributed computing system on its own (i.e. DIRAC is a complete middleware). At the same time, DIRAC provides the means to connect, among others, to most common distributed computing services in scientific world:

- VOMS: for user registration
- BDII: for resource discovery
- Resource allocation with Cluster, Grid, Cloud and Volunteer interconnecting common interfaces in a transparent manner, for example:
    - Cluster batch systems: Torque, Condor, SLURM, LSF
    - Grid: gLite WMS, CREAM CE, ARC CE
    - Cloud: OpenStack, OpenNebula, EC2
    - Volunteer desktop computing: BOINC
- Storage management connecting grid and other storage services:
    - SRM: for accessing storage resources
    - Other storage resources like Gridftp or DIRAC vanilla Storage Element
    - LFC: optional service for replica location
    - FTS: for file transfer service
    - DIRAC File Catalog (DFC): Replica location, metadata and provenance

DIRAC framework [2] has a core module and different extensions modules providing additional services that can be eventually installed like Web (for the DIRAC web portal) or VMDIRAC (for DIRAC connection to cloud), or enhanced features for particular communities (for example LHCbDIRAC, BESDIRAC or BelleDIRAC). DIRAC WebApp extension module allows developing desktop like web applications and specific services for communities.

DIRAC provides the high level functionalities in several API for different purposes. Furthermore, third-party applications can be developed powered by DIRAC services using native Python API. The Python API is also providing HTC PaaS functionality to simplify distributed computing in the client side. For this purpose, DIRAC python client is deployed in the pilot job [3] on cluster and grid infrastructures or in the Virtual Machine (VM) [4] on Cloud

and Volunteer infrastructures. This PaaS architecture moves part of the logic from the service side to the client, enabling better scalability of the systems. Hereby, such intelligent agents close to the particular distributed application alleviate the complexity of central algorithms and methods. DIRAC also provides a programming language agnostic REST interface [5]. Currently, REST interface can be used to submit and monitor jobs, check accounting information or query the file catalogue from external systems. To access the complete DIRAC web functionality one can use DIRAC *JavaScript* utilities, which can also be embedded in third-party portals.

The rest of this paper is organized as follows, Section 2 describes the underlying architecture of DIRAC APIs: Python, Web and REST, including the main functionalities used in several developing scenarios: applications on top of DIRAC, HTC PaaS wrappers at the client side, Web applications and REST applications. The paper concludes in Section 3 including major work in progress.

2. DIRAC API Architectures

DIRAC is a platform for scientific computing providing community management tools for group management with user privileges, community wide policies and automated synchronization with VOMS services.

It also supports a Workload Management System adapting the scheduling to user priority, community policies and to the status of the resources. A third-party dynamic information system, like BDII, can be integrated in DIRAC automating proactive actions to different resource status. Community wide data management policies are also contemplated in DIRAC.

There are some advanced functionalities to simplify scientific computing:

- DIRAC has a bulk submission method with parametric jobs.
- Similar job can be predefined in job launch-pad templates.
- Data drive workflows can be defined using its Transformation System.
- Data Management including metadata to identify user data, replica location and placement, and provenance to track the source of the data.
- Resource usage accounting for providers, communities, groups and users. Common accounting metrics for CPU, Jobs, Wall-time, Storage or Transfers.

DIRAC platform provides access to different combinations of these functionalities depending on the selected API.

2.1 DIRAC Python API

DIRAC API is encapsulated in several Python classes designed to access a large fraction of the functionalities. Using the API classes it is easy to write small scripts or applications to manage user jobs [3] and data processing with metadata [6] and provenance [7] management for HTC. These scripts can be used in the client side as PaaS or encapsulated in the server side, for example in a VRE. A DIRAC client installation[1] is required in order to use this API. An UML diagram of the DIRAC API classes is shown in Figure 1.

[1] http://diracgrid.org/files/docs/UserGuide/GettingStarted/InstallingClient/index.html

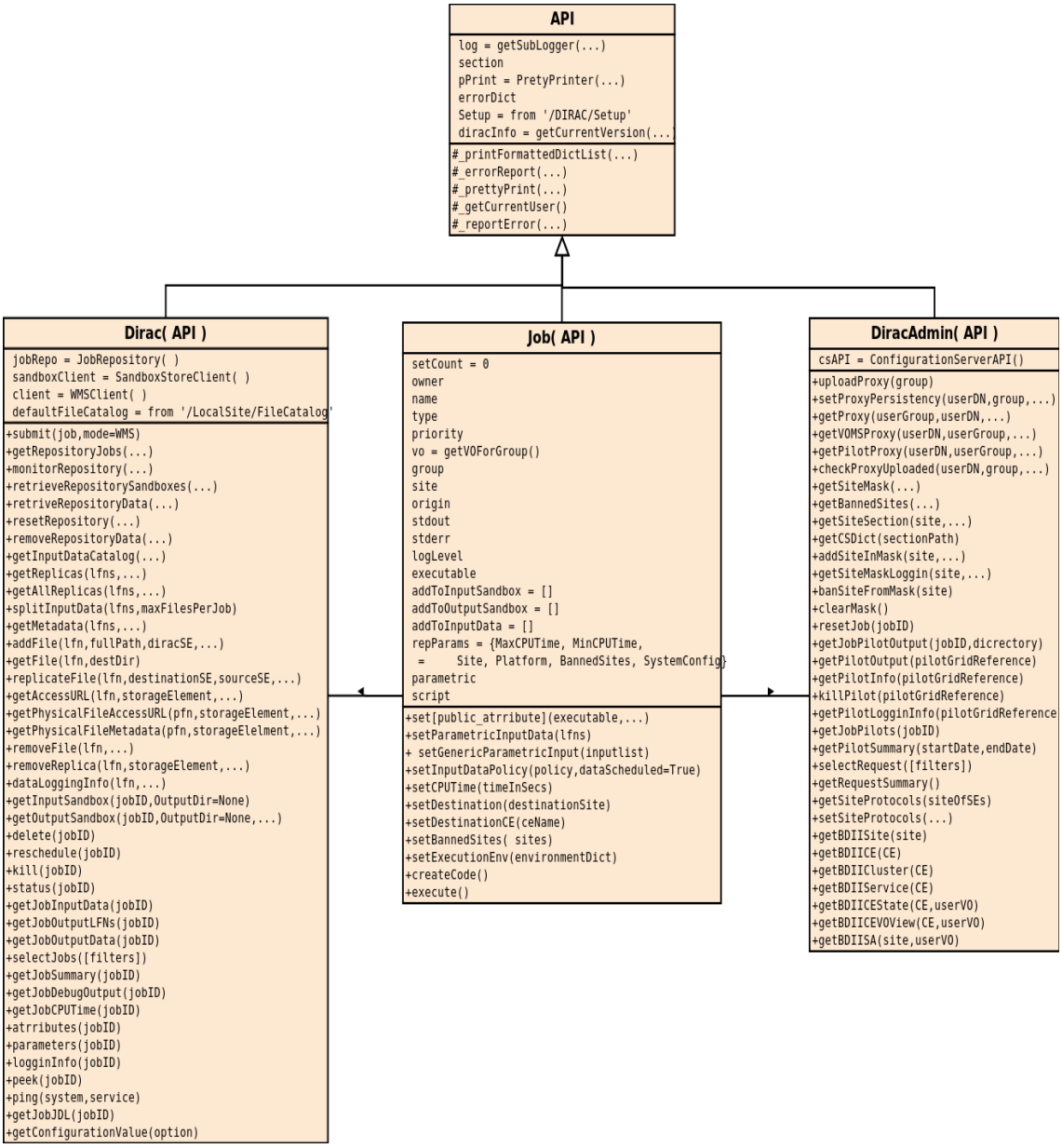

Figure 1 - DIRAC API, UML Diagram -

On the top of Figure 1, *API* supper-class defines the common environment of the three other inherited classes. There are separated classes for administrator and, user that can be used with the corresponding privileged proxy or simple user proxy. *Job* instances are managed by *Dirac* and *DiracAdmin*.

Job API in the centre of Figure 1 is a job representation [8]. The classical JDL parameters are defined with *set<public_attribute>()* functions like. It is possible to specify input data by LFN to be used as a parameter in a parametric job with *setParametricInputData* and to define a generic parametric job with *setGenericParametricInput*. If the parametric job requires input sandbox files they can be specified with *setParametricInputSandbox.* Function

setInputDataPolicy allows selecting file access policy, 'Download' or 'Protocol'. This requires that the module locations are defined in the Configuration. Previously to the job execution it is possible to filter the computing sites by *setDestination, setDestinationCE,* and *setBannedSites.* Running variables can be defined with *setExecutionEnv* providing a dictionary of key, value pairs. The standard application environment variables are always set, so this is intended for user variables only. Internal workflows within a DIRAC job can be used with *createCode*. The job can be locally executed without submission, using *execute()* function.

Dirac API has two function categories for job and data management. Job management includes the job submission of a previously instantiated job object and a set of Job operations (*delete, reschedule, kill, status*), *jdl* operations (getting *attributes, parameters* and the *get<JDL_element>*. Other job operations are *selectJobs* depending in the *filter* and the operation *peek* attempts to return standard output for a given job if available. The standard output is periodically updated from the executing node via the application Watchdog. Group of jobs can be aggregated in a *jobRepository* object to manage bulk operations in *<operation>Repository* functions like. Data management does not require a previous job object. These HTC data operations are used from external applications, like VRE or Science Gateways servers, and also at the client side within the working node or VM which is processing the job. . When a catalogue is provided (DFC or LFC), it is possible to operate with it to obtain replica information using a Logical File Name (LFN), using *getReplicas* to consider the unbanned storages or also including banning sites with *getAllRepllicas*. Large set of input data can be assigned to multiple jobs with *splitInputData*. Other *File* operations with LFN are *addFile, getFile, replicateFile, removeFile, removeReplica.* Having a LFN it is possible to get grid storage URL and associated storage protocol with *getAccessURL*, which can also be obtained from a Physical File Name (PFN) with *getPhysicalFileAccessURL.* When using DFC standalone or in combination with other catalogue like LFC, it is possible to use *metadata* operations, such as *getMetadata* and *getPhysicalFileMetadata* to filter file queries and latter do bulk job submissions with selected input files.

DiracAdmin API is used to some restricted resources operations. Group properties are assigned in the Configuration to define privilege access to operations and resources. There is a set of operations for some resource management like pilots, proxies, site mask, monitoring request, define and obtain a site storage protocols and poll BDII information system. This API is designed for DIRAC extensions, rather than external applications.

2.2 DIRAC Web API

DIRAC Web extension has been fully re-written in the last 18 month, it is the third DIRAC web interface version in ten years, taking advantage of previous experience, new tools and technologies.

New web framework design goals are:

- Improve modularity and extendibility
- Allow usage in a single browser tab
- Keep a similar look and feel
- Improve usability

- Obtain a robust framework

The resulting web developing framework is shown in Figure 2. The server side is a Tornado web service running DIRAC client tools. Tornado allows web service scalability including the use of multiple tornado instances when necessary.

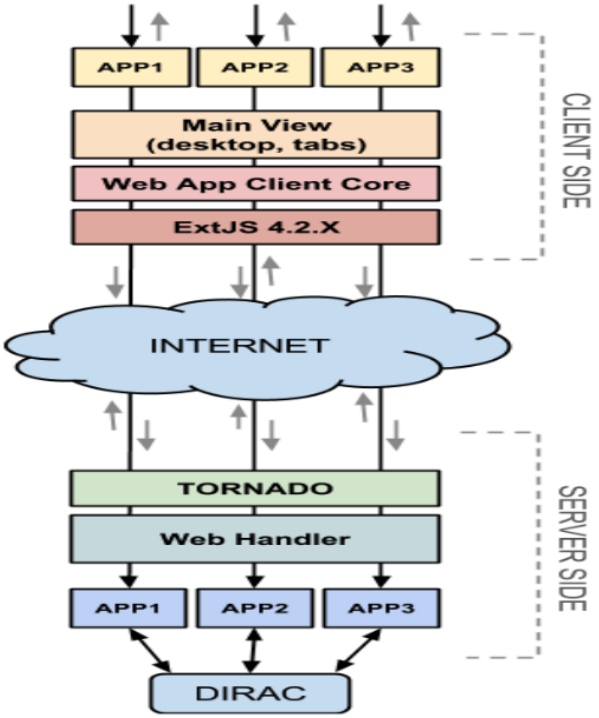

Figure 2 - New DIRAC Web Architecture –

The client side is *JavaScript* with heavy use of ExtJS 4.2.X, embedded in HTML pages, with CSS web page formats and Google charts. *Main View* of a Web application is composed of a desktop within the browser. . The application can be designed inside a DIRAC extension but can also be placed in third party web portals, for example, CTA portal obtains job monitoring and accounting plots. A functional perspective of this architecture is shown in Figure 3 focused in the WebApp CORE of the server side. WebAppDIRAC extension is the web front-end extension. Core objects support a common framework for DIRAC Web or other web applications based in DIRAC, providing:

- One browser tab per user ID
- General functionality and widget toolbox
- Navigation and settings
- Integrates several views:
    - Desktop like display (Default)
    - Tabs like display (being developed by LHCb)
- User Profile Management (save, load, share, etc. states)

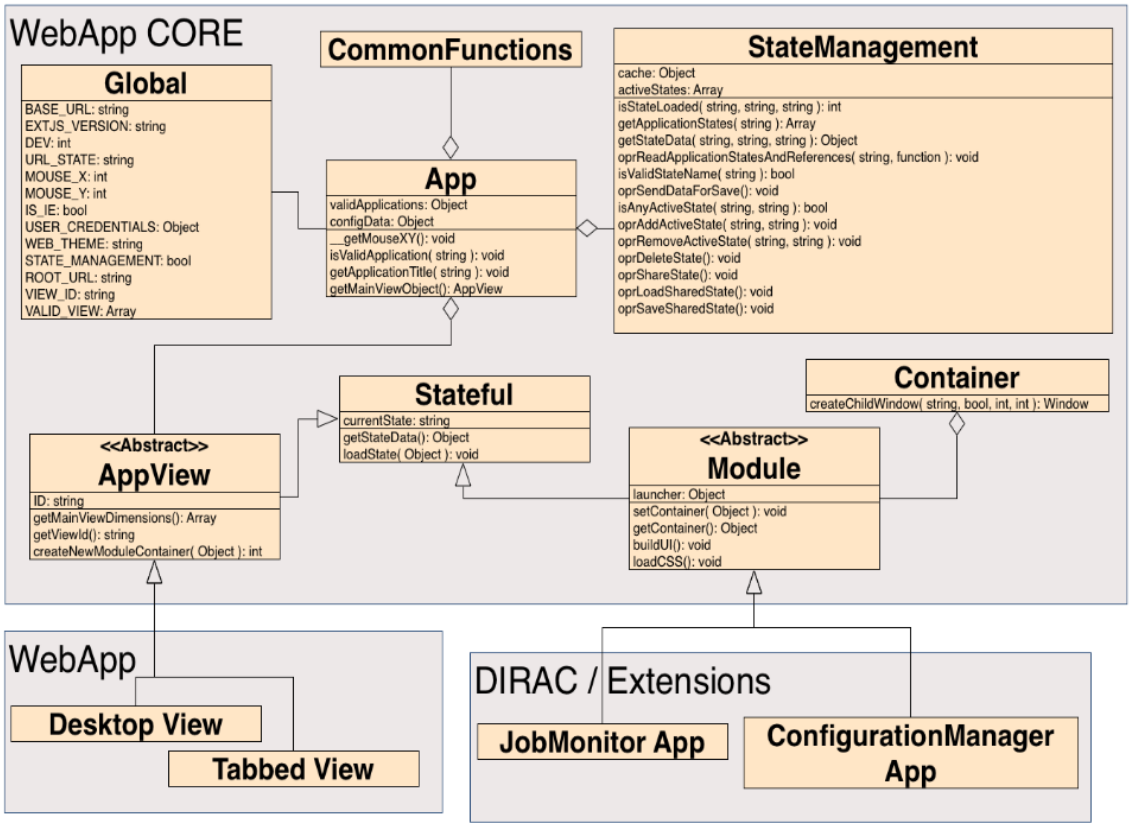

Figure 3 - DIRAC Web API, UML Diagram

App is the basic class in the architecture. It configures and creates web applications as JavaScript clients with synchronous or asynchronous handling mechanism for browser events. Each application server side logic is implemented in one Python file. The name of the file is formed by appending the word Handler to the name of the application. This file defines a Python class responsible for all server side functionality. A method, to be accessible in the application class, needs the prefix "web_" to the name of the method. In order to send back response to the client, there is the *write* method of the *WebHandler* class. If response is of type dictionary, then the dictionary is converted to JSON string before it is sent back to the client. The Tornado server handles all requests one-by-one meaning that the server does not handle the next request until the current one is finished. This mechanism becomes a bottleneck if one request lasts longer and increases the response time for each subsequent request waiting in the server queue. To mitigate the issue, the server provides a way to asynchronously handle client requests.

Once the web application services are defined, client side has a clear recipe to provide specific information for the user. Client side consists of files needed for rendering the UI and communicating with the server side. These files are placed in *WebApp/static/<module name>/<application name>* with several folders:

- *build*: this folder contains the compiled version of the *JavaScript* files contained in the *classes* folder
- *classes*: this folder contains the *JavaScript* file that defines the main ExtJS class representing the application on the client side. A mandatory file <application name>.js:

contains the main ExtJS class representing the application on the client side. The name of the file must have the same name as the application we want to build.

- *css*: this folder contains all the css files specific to this application. A mandatory file with name <application name>.css: contains the *css* style needed by the components of the application.
- *images*: this folder contains all the specific images and icons needed by this application.

When extending the base class within the *JavaScript* file, there are some mandatory methods to be implemented within the derived class:

- *initComponent:* this method is called by the constructor of the application. This method sets up the title of the application, its width and height, its maximized state, starting position on the screen and the icon css class. Here it is suitable to define the layout of the entire application.
- *buildUI:* this method is used to build the user interface. Usually this is done by instantiating ExtJS widgets. These instances are added to the application in a way prescribed by the layout defined in the initComponent method. This method is called after all the css files regarding this application have been successfully loaded.
- *getStateData*: The DIRAC web framework provides a generic way to save and load states of an application. This method is not mandatory, and it can be overridden by a new implementation in the application class. Whenever the user saves an application state, this method is called in order to take the data defining the current state of the application. The data has to be a *JavaScript* object.
- *loadState(data)*: This method is being called to load a state. As an argument the framework provides the data that have been saved previously for that state.

There is a DIRAC how-to develop new web applications[2] describing further technical details, examples, and predefined widgets[3]. It is possible to inherit from the available web applications and every application can run independently. In this way everyone can produce a fast set of web applications for his particular purposes. DIRAC default web applications are: *AccountingPlot, ActivityMonitor, ConfigurationManager, ExampleApp, FileCatalog, JobLaunchpad, JobMonitor, Notepad, PilotMonitor, PilotSummary, ProxyUpload, RegistryManager, RequestMonitor, SystemAdministration, TransformationMonitor.* There are further applications in DIRAC extensions like LHCbDIRAC.

2.3 DIRAC REST API

DIRAC REST API provides a language neutral interface for third party clients connecting to DIRAC. This API follows the REST style over HTML, using JSON as the serialization format. All requests to the *REST* API are *HTTP* requests powered by Tornado web application architecture, which can directly connect to DIRAC web service applications or to a REST server side application following similar architecture. OAuth2 is used as the credentials delegation mechanism to the applications.

[2] https://github.com/DIRACGrid/WebAppDIRAC/wiki/How-to-create-new-web-application-within-the-DIRAC-web-framework

[3] https://github.com/DIRACGrid/WebAppDIRAC/tree/integration/WebApp/static/core/js/utils

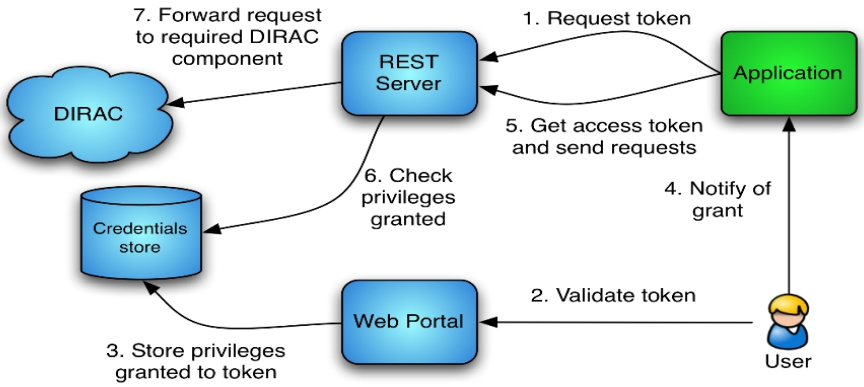

Figure 4 - REST Authentication Process Flowchart

Figure 4 shows the authentication process. The server, that exports the *RESTful API*, also generates and exchanges *OAuth* tokens. Moreover, a DIRAC Web Portal module exists that allows users to validate tokens, where different privileges and life times can be granted. A credential store is provided where tokens and privileges are kept while tracking all operations done to and with any token. All RESTful servers check with the credential store the validity of the received tokens.

Once the client has a valid access token, it can use the REST API. All data sent or received will be serialized in JSON. Main REST interface is *JobManagement*, with the following interface:

- GET /jobs: Retrieve a list of jobs matching the requirements.
- GET /jobs/<jid>: Retrieve info about job with id=*jid*
- GET /jobs/<jid>/manifest: Retrieve the job manifest
- GET /jobs/<jid>/inputsandbox: Retrieve the job input sandbox
- GET /jobs/<jid>/outputsandbox: Retrieve the job output sandbox
- POST /jobs: Submit a job. The API expects a manifest to be sent as a JSON object. Files can also be sent as a multipart request. If files are sent, they will be added to the input sandbox and the manifest will be modified accordingly.
- DELETE /jobs/<jid>: Kill a job. The user has to have privileges over a job.

A job accounting REST interface connects the Job Accounting service to get job statistics. Another REST interface connects the File Catalog service with REST wrapper (BasicFC) to launch metadata queries and manage JSON request and reply formats. REST API is used to connect REST clients like the ones embedded in the CTA Scientific Gateway or Web applications like DIRAC4Androi extension, written in java.

3. Conclusions

DIRAC engine can be used in several distributed system developing scenarios such as applications on the top of DIRAC, HTC PaaS wrappers at the client side, Web applications and REST applications. DIRAC APIs in Python, WebApp and REST support modular and extendible design based on a flexible framework. Distributed applications powered by DIRAC

overcome the digital divide for the big science and the long tail of science, taking advantage of the synergies, allowing cost-savings, encouraging best practices and sharing experience. DIRAC project is an open source framework and the corresponding shared activities for development, training or consultancy and support services.

There is a work in progress to offer more functionality in the APIs. Main working lines are analysing enhanced HTC capabilities, VRE requirements for workflow pipelines and additional PaaS utilities in the server and client side.

References

[1] Andrei Tsaregorodtsev, Ricardo Graciani, Adrian Casajus, Víctor Méndez, Federico Stagni, Vanessa Hamar, Matvey Sapunov. Status of the DIRAC Project. Journal of Physics Conference Series (Computing in HEP). Vol 396. Issue 032107, 2012

[2] Adrian Casajus and Ricardo Graciani in b. of Lhcb Dirac Team. DIRAC distributed secure framework, Journal of Physics Conference Series, Vol 219. Issue 4, 2010

[3] Adrian Casajus and Ricardo Graciani and Stuart Paterson and Andrei Tsaregorodtsev in b. of Lhcb Dirac Team. DIRAC pilot framework and the DIRAC Workload Management System. Journal of Physics Conference Series, Vol 219. Issue 6, 2010

[4] Vícitor Méndez and Adrian Casajús Ramo and Víctor Fernández Albor and Ricardo Graciani Diaz and Gonzalo Merino Arévalo. Rafhyc: an Architecture for Constructing Resilient Services on Federated Hybrid Clouds. Journal of Grid Computing, Vol 11. Issue 4, 2013

[5] Adrian Casajus Ramo, Ricardo Graciani Diaz, and Andrei Tsaregorodtsev. DIRAC RESTful API. *Journal of Physics: Conference Series*. Vol. 396. Issue. 5. IOP Publishing, 2012.

[6] A C Smith and A Tsaregorodtsev. DIRAC: reliable data management for LHCb. Journal of Physics Conference Series, Vol 119. Issue 6, 2008

[7] Bargiotti, Marianne, and Andrew C. Smith. DIRAC Data Management: consistency, integrity and coherence of data. *Journal of Physics: Conference Series*. Vol. 119. Issue 6. IOP Publishing, 2008.

[8] Pacini, F., & Kunzt, P. (2006). Job description language attributes specification. EGEE Project.