EPJ Web of Conferences **245**, 03020 (2020) https://doi.org/10.1051/epjconf/202024503020
CHEP 2019

DIRACOS: a cross platform solution for grid tools

Marko Petrič[1,*], *Christophe* Haen[1], and *Benjamin* Couturier[1]

[1]CERN, Geneva 23, Switzerland

Abstract. DIRACOS is a project aimed to provide a stable base layer of dependencies on top of which the DIRAC middleware is running. The goal was to produce a coherent environment for grid interaction and streamline the operational overhead. Historically the DIRAC dependencies were grouped in two bundles; Externals containing Python and standard binary libraries, and the LCGBundle which contained all grid-related libraries (gfal, arc, etc). Such a setup proved difficult to test and hindered agile development. DIRACOS solves the binary incompatibility that was caused by using a python version newer than the native system one (SLC6). It is spawned from a single list of required packages from where we pull all dependencies down to the level of glibc using SRPMs. With such an approach we can provide a uniform set of packages for our clients, servers, and several platforms. It is an extendible setup with a DevOps development cycle in mind. The core build functionality of DIRACOS is based on Fedora Mock. DIRACOS introduces its own grammar to handle specific cases, and it also allows patching (some SRPMs require tweaking, which the user can do by providing a diff) as well as routines for pre/post/instead actions of compilation. With this approach DIRAC was able to provide a single bundle for clients and servers that is reliable, flexible, easy to test and relatively small (250 MB). It allows for a smooth transition from SLC6 to CC7 and provides a clear roadmap for possible extensions of DIRAC to a wide variety of platforms.

1 Introduction

DIRAC interware is a software framework for distributed computing providing a comprehensive solution for distributed computing to a myriad of different communities [1–3]. It provides a layer between the users and the resources offering a common interface to a number of heterogeneous providers and is extensible horizontally and vertically (see figure 1). To accommodate all these features and extensibility, DIRAC comes with many external libraries and dependencies. They can be categorised into:

1. Python libraries
2. Middleware libraries
3. Server side tools

All these dependencies have previously been managed by two independent packages called Externals and LCG Bundle.

*e-mail: marko.petric@cern.ch

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

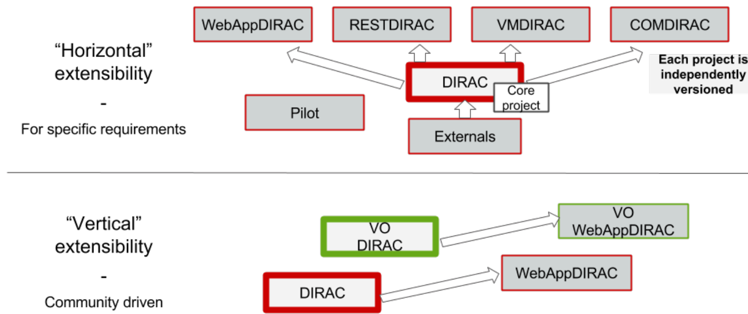

Figure 1. Different possibilities to extend DIRAC.

1.1 The Externals

The Externals package contains python and standard binary libraries. It was pre-compiled for several platforms (SLC6, CC7...). Depending on the usage, there was a version to be deployed on servers (190 MB) and another version for usage with the client application (13 MB). The Externals package had a lower frequency of changes compared to the Bundle, and was maintained by DIRAC.

1.2 The LCG Bundle

The LCG Bundle contained libraries from the grid world (gfal, arc...) and even some duplicates with respect to the Externals. They were compiled only for two `glibc` versions, 2.12 (SLC6) and 2.17 (CC7), and the bundle did not differ for the server or client installations (80 MB). The bundle was created by cherry picking binaries from existing LCG-Application-Area releases produced by CERN EP-SFT.

1.3 The issue

Such a setup of handling dependencies posed a problem, as DIRAC does not have any influence on the release cycle and inclusion of specific versions in the LCG-Application-Area releases. Problems could also arise from collisions of versions of binaries in the Externals. Furthermore, such a setup proved to be very difficult as one has to check the interplay of 3 different packages (Externals User vs. Externals Server vs. LCG Bundle). This hinders a fast release cycle, and ties DIRAC to the LCG-Application-Area release cycle. An additional obstacle is that the provisioning of these packages requires manual intervention and there is no extension mechanism to support the native extendibility of DIRAC. Apart from this maintenance and release management issue, the two packages also have several technical issues:

1. Different binaries for different platforms
2. The versions used in the build are not system versions of binaries
3. LCG-Application-Area releases depend on the compiler version used to produce the bundle
4. There is a binary incompatibility when a python version different than the one from the system is used

5. The used stdlib is without C++11 support

2 DIRACOS

We have designed DIRACOS to overcome all the above explained issues and weaknesses. A main design feature of DIRACOS is that it is portable, which means that the same binary is used for several platforms. The development currently supports SLC6 and CC7, however tests have shown that it can be used also on different platforms. An additional important feature of DIRACOS is that it is easily extendable, which enables each community to extend the base package for their particular use case[4].

2.1 Building

All packages contained in DIRACOS are built from a single package list, which ensures a consistent build. All the the binaries are recompiled from SRPMs and all the dependencies are resolved and rebuilt down to the level level of `glibc`. Through such a procedure we produce a monolithic package for server and client use, which is usable on different platforms. This approach also solves the binary incompatibility of using a python version newer than the native system. The build is based on Fedora Mock [5] and yum repo, which creates `chroots` and builds packages in them. These tools are the base for all Fedora builds and underlie Koji [6] and Copr [7].

2.2 Implemented a grammar

Developers are often faced with special use cases. To cope with these situations, DIRACOS implemented a grammar. The grammar enables users to apply patches to RPM `spec` files by adding diff files to the patch folder. The system thereupon automatically applies the patch. The grammar gives users the possibility to use routines for pre, post and instead actions (figure 2).

```
{
  "rpmBuild": {
    "opt1": 1,
    "package Groups": [
      {
        "name": "pkgGroup1",
        "opt1": 2,
        "packages": [
          {
            "name": "pkg1",
            "opt2": "x"
          },
          {
            "opt1": 3
          }
```

Figure 2. Example of DIRAC Grammar.

2.3 Easy build procedure

DIRACOS provides tools for easy generation. To start the procedure one first installes dependencies:

```
yum install -y mock rpm-build fedora-packager createrepo python-pip
pip install diracos
```

To compile the bundle:

```
dos-build-all-rpms config/diracos.json
```

A command is provided to generate a `requirements.txt` with all python packages and accompanying versions:

```
dos-fix-pip-versions config/diracos.json
```

A command is provided to compile a python module:

```
dos-build-python-modules config/diracos.json
```

To finalise a build one pulls all dependencies and creates a tar file by executing

```
dos-bundle config/diracos.json
```

It is also very easy to extend the basic DIRACOS by using a custom `requirements.txt` file and a small json configuration

```
{
  "extensionName": "LHCb",
  "diracOsVersion": "v1r4",
  "version": "master",
  "pipRequirements": "lhcb_requirements.txt"
}
```

coupled with the execution of the command

```
dos-build-extension lhcbdirac.json
```

2.4 Testing

DIRACOS is equipped with a continuous integration suite. For each merge a posterior build is executed. A part of the integration suite is also a check for broken symbolic links and whether there are any absolute symbolic links to ensure relocatability. There is a built-in mechanism to handle broken links and whitelist certain libraries. If all these steps can be fulfilled, DIRACOS is initialised in a vanilla container and we test whether all python packages can be imported, and at the same time whether we can run command line interface tools of constituent packages like gfal. If all these tests pass, we try to run the complete set of integration tests of DIRAC. This enables us to quickly check the effect of version changes on the operations of DIRAC. A detailed flow of the testing procedure can be seen in figure 3.

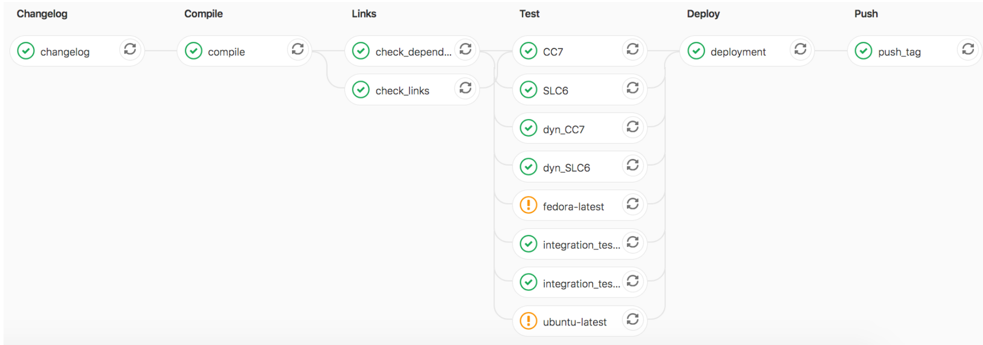

Figure 3. The testing flow of DIRACOS.

The release procedure is fully integrated in the continuous integration. It is triggered when a branch with a name starting with *rel-** is merged. Firstly all the PR information with the release notes is aggregated since the last release. This is followed by a whole pass of the testing infrastructure. If successfully passed, all python package versions are frozen and added back to the repository together with the releases notes. At the end the system tags the repository and uploads a tar file of the release. A release is published on GitHub with all the information about the changes this release contains plus a comprehensive list of all package versions (see figures 4 and 5).

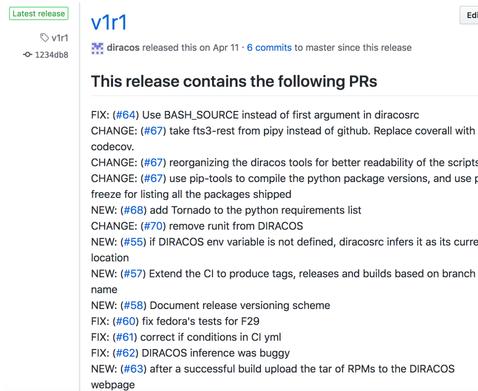

Figure 4. Display of aggregated changes in a GitHub release.

Included versions of packages

===== RPM packages ====

autoconf-2.63-5.1.el6.noarch.rpm
automake-1.11.1-4.el6.noarch.rpm
boost-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-date-time-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-devel-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-filesystem-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-graph-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-iostreams-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-math-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-program-options-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-python-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-regex-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-serialization-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-signals-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-static-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-system-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-test-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-thread-1.41.0-28.el6.py27.usc4.x86_64.rpm
boost-wave-1.41.0-28.el6.py27.usc4.x86_64.rpm

===== Python packages ====

astroid==1.6.6
atomicwrites==1.3.0
attrs==19.1.0
autopep8==1.3.3
backports-abc==0.5
backports.functools-lru-cache==1.5
backports.shutil-get-terminal-size==1.0.0
certifi==2019.3.9
chardet==3.0.4
CMRESHandler==1.0.0
codecov==2.0.15
configparser==3.7.4
coverage==4.5.3
cx-Oracle==7.1.2
cycler==0.10.0
decorator==4.4.0
docopt==0.6.2
docutils==0.14
elasticsearch==6.3.1
elasticsearch-dsl==6.3.1

Figure 5. Listing of versions of libraries that are contained in a DIRACOS release.

3 Summary

Great progress has been made since the start of the DIRACOS project. The project has reached production level quality and is the default feature used in DIRAC since release v7r0, replacing the previous two package options. The size has been reduced to 230 MB in the last release. It has been demonstrated that with DIRACOS the release cycle can become more agile, and enables an easier reaction to rapid changes when necessary. The repository of the project is hosted at https://github.com/DIRACGrid/DIRACOS.

References

[1] F. Stagni et al., Journal of Physics: Conference Series **898**, 092020 (2017)
[2] A. Tsaregorodtsev et al., Journal of Physics: Conference Series **119**, 062048 (2008)
[3] F. Stagni et al., *DIRACGrid/DIRAC* (2019), https://doi.org/10.5281/zenodo.1451647
[4] C. Haen, M. Petric, B. Couturier, *DIRACGrid/DIRACOS* (2020), https://doi.org/10.5281/zenodo.3710832
[5] *Fedora mock*, (accessed: 2020-03-14), https://fedoraproject.org/wiki/Using_Mock_to_test_package_builds
[6] *Fedora koji*, (accessed: 2020-03-14), https://koji.fedoraproject.org/koji/
[7] *Fedora copr*, (accessed: 2020-03-14), https://copr.fedorainfracloud.org/