EPJ Web of Conferences **214**, 05020 (2019) https://doi.org/10.1051/epjconf/201921405020
CHEP 2018

Building, testing and distributing common software for the LHC experiments

Javier Cervantes Villanueva[1], *Gerardo* Ganis[1,*], *Dmitri* Konstantinov[2], *Grigorii* Latyshev[2], *Pere* Mato Vila[1], *Patricia* Mendez Lorenzo[1,**], *Rafal* Pacholek[3], and *Ivan* Razumov[2]

[1]CERN, Switzerland
[2]NRC Kurchatov Institute - IHEP, Protvino, Russia
[3]AGH University of Science and Technology, Poland

Abstract. Building, testing and deploying of coherent large software stacks is very challenging, in particular when they consist of the diverse set of packages required by the LHC experiments, the CERN Beams Department and data analysis services such as SWAN. These software stacks include several packages (Grid middleware, Monte Carlo generators, Machine Learning tools, Python modules) all available for a large number of compilers, operating systems and hardware architectures.
To address this challenge, we developed an infrastructure around a tool called `lcgcmake`. Dedicated modules are responsible for building the packages, controlling the dependencies in a reliable and scalable way. The distribution relies on a robust and automatic system, responsible for building and testing the packages, installing them on CernVM-FS and packaging the binaries in RPMs and tarballs. This system is orchestrated through Jenkins on build machines provided by the CERN Openstack facility. The results are published through user-friendly web pages.
In this paper we will present an overview of these infrastructure tools and policies. We also discuss the role of this effort within the HEP Software Foundation (HSF). Finally we will discuss the evolution of the infrastructure towards container (Docker) technologies and the future directions and challenges of the project.

1 Introduction

Modern physics experiments require complex software stacks to build the experiment specific applications. Stability and reproducibility usually imply a conservative approach in the choice of the main components; however, the constant need to integrate new developments and versions to improve performances and usability makes the provision of coherent large software stacks a dynamic and challenging task. In this paper we present and discuss the way the CERN EP-SFT [1] group has addressed this task for the needs of ATLAS [2], LHCb [3], SWAN [4], FCC [5] and the CERN Beams Department [6].

The paper is organized as follows. In the rest of this section we will introduce some key concept and related terminology. In the next section we discuss the `lcgcmake` tool,

*e-mail: gerardo.ganis@cern.ch (corresponding author)
**e-mail: patricia.mendez@cern.ch (presenter)

based on CMake [7] and used to build the required packages. We then describe how this tool is integrated in a build and deployment infrastructure based on Jenkins [8], and the way information about the package content of a given release version is provided. Finally we mention the current and future consolidation and development work.

1.1 The LCG stacks

Figure 1 gives a schematic overview of the various components of the software stack of an experiment.

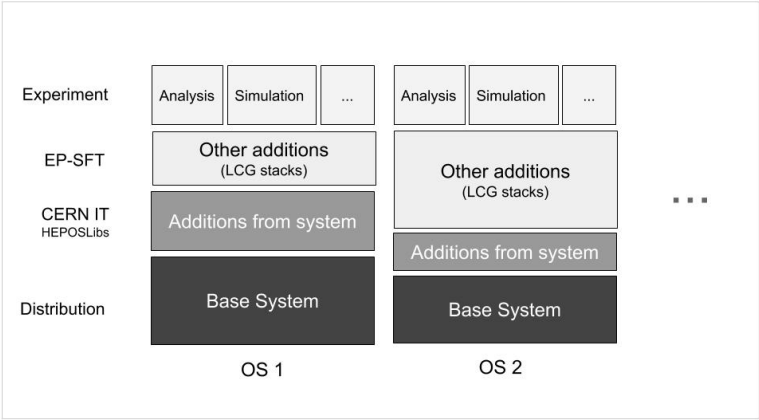

Figure 1. Overview of the various components of an experiment software stack.

The experiment applications determine the package content of the entire stack. Some of these packages may be available in the official distribution repositories of the given operating system (OS); these represent the system prerequisites and in the figure these are the two blocks at the bottom [1]. The provision of the *LCG stacks*, which stand between the experiments applications and the system provided components, is the subject of this paper.

The LCG stack is made of packages. A package can be of three types:

- `contrib`: these are utility packages upgrading the equivalent packages provided by the system; typical examples are compilers or, for some systems, CMake;
- `projects`: these are packages developed in CERN EP-SFT or related CERN IT projects; examples are ROOT [10], Geant4 [11], COOL/CORAL [12];
- `externals`: this category comprises all the remaining packages. Three sub-categories are singled-out, mostly for historical reasons:
  - `gridexternals`: these sub-category includes packages providing functionality related to the Grid;

[1] In the case of the LCG stacks, the system pre-requisites are provided by a meta package called HEPOSLibs [9] which will not be discussed any further.

– `pyexternals`: these sub-category contains purely Python external packages;
– `generators`: these sub-category contains physics generators; the particularity is that more versions of the same generator can be part of stack; support for multi-version will be discussed later on.

The system currently manages about 400 packages; these packages are mostly written in C++ and Python, though there are still C and Fortran packages, in particular among the generators.

1.2 Type of builds: *releases* and *nightlies*

To provide reproducibility, exact control of the content of a given stack is required. To achieve that, a major version number and a tag is assigned to the set of *package*, *version* which constitutes an LCG stack. A tagged LCG stack is called a *release*. At the time of writing, the major version is `94` and the corresponding tag is `LCG_94`.

A particularity of the LCG stacks is that the change in the global major number is driven by new releases of the ROOT package, one of the project packages on which almost all the experiment applications depend on; as an example, `LCG_94` is based on ROOT `v6-14/04`.

In order to test future releases, two development builds are provided, which differ by the ROOT version included. For historical reasons, these are called `dev3` and `dev4` and have the following composition:

- `dev3`: head of ROOT master; latest versions of validated packages;
- `dev4`: head of patches branch of latest ROOT tagged release, e.g. `v6-14-00-patches`; latest versions of validated packages.

The development builds `dev3` and `dev4` are provided on daily basis and called *nightlies* (they are built during nights); the builds are kept for seven days.

1.2.1 Python 3 builds

The release tag is also used to indicate a particular specialisation of a build. The notable example is the case of builds with Python 3, the artefacts of which cannot be mixed with those of the default build for which Python 2 is used. Python 3 builds are flagged with the suffix `python3` - e.g. `LCG_94python3`; de facto, these are considered as a different platform (see Sect. 1.4).

1.3 The life-cycle of a new package

A new package is initially built on a local node to determine the relevant configuration and build options. A first integration build, flagged *experimental* is then attempted and, if successful, the package is included in the development builds, to appear in the *nightlies*. The new package will then be included in the next *release*.

1.4 Platform concept

In order to keep up the needs of the experiments in terms of OS, architecture, compiler, debug, instruction sets, etc. the concept of *platform* is introduced. The platform identifies a given build configuration following the scheme [13]

Architecture-OS-Compiler-BuildType

where the different components carry the following meaning:

- *Architecture* indicates the computer architecture for which the build is made. Most of the builds are done for `x86_64` or extension of that, e.g. `x86_64+avx2+fma` . Support for `arm64` is still experimental and part of the future work. 32-bit architectures are not required anymore and not built by default;
- *OS* denotes the operating system. The reference OSs are the Linux flavours proposed by CERN, which, at the time of writing, are Scientific Linux CERN 6 (`slc6`) and CERN CentOS 7 (`centos7`). Ubuntu builds are provided for single users, not necessarily from the customer experiments; these builds provide a test of the entire procedure on newer distributions and to anticipate problems;
- *Compiler* brings information about the compiler name and its version. The choices in terms of compilers are driven by the experiment's needs. The set of compilers in use could be split into two different categories: production-ready compilers such as GCC 6.2 and GCC 7 [2], and more recent versions or compilers under testing such as GCC 8 or Clang 6.0.0 . The term `native` refers to the compiler coming with the system and it is only used when it is modern enough, as usually on the latest Ubuntu OSs;
- *BuildType* denotes whether the build is a debug build, an optimized build, and any other special setting. *Debug* (`dbg`) and *optimized* (`opt`) are treated as separate builds. Debug build are produced for all packages, though this may not be strictly required and will be reviewed in the future.

Table 1 shows the combinations currently managed. Not all the combinations are built; for example, the `LCG_94` release has been provided for 14 combinations for each Python flavour; the development builds contain a similar number of combinations, focusing more on combinations which need validation and testing, such as those involving recent compilers.

Table 1. Platform combinations: choice currently supported. In parenthesis those more experimental not used for releases. The list of platforms actually available for a given release or nightly build are provided by the LCGInfo web site at `http://lcginfo.cern.ch`

Architecture	*OS*	*Compiler*	*Build Type*
`x86_64`	`slc6`	`native`	`opt`
`x86_64+avx2+fma`	`centos7`	`gcc62`	`dbg`
(`arm64`)	`ubuntu16`	`gcc7`	
	`ubuntu18`	`gcc8`	
	(`mac`)	`clang60`	

1.5 Deployment and packaging

One of the relevant aspects relates to the way the additional binaries are provided. The main distribution vector for the LCG stacks is CernVM-FS [14], a read-only, cache-aggressive network file system optimized for software distribution and widely used at LHC. The project manages two CernVM-FS repositories for this purpose:

`/cvmfs/sft.cern.ch`	contrib packages, releases, views
`/cvmfs/sft-nightlies.cern.ch`	nighlies development builds

[2] The meaning of the major number in GCC releases changed with version 7: the minor did become the major, so that it is not necessary anymore to specify the minor when specifying the GCC version.

The main reason to have two different repositories is the different life cycles of the information included: the main repository `/cvmfs/sft.cern.ch` contains reference information; the other repository contains information being recycled and garbage collected after one week. To give a common view to all the deliverables, releases and development builds, symlinks to the relevant nightlies directories are created under the main repository, so that it is possible, de facto, to work only with the main repository.

Views

To simplify the setup of a given set of packages, either a release or a nightly, as well as runtime lookups, the concept of *view* has been introduced. A view is a path that contains all what is required to run the chosen release or nightly as a large global meta-package, with a Linux-system-like organisation of the packages and files provided by a release or nightly. For the LCG stacks, views are created on CernVM-FS under `/cvmfs/sft.cern.ch/lcg/views` for each release and nightly build sym-linking the relevant files, for example under the common `bin`, `lib`, `etc`, `include` directories. To use a view, the user only needs to source the relevant `setup.{sh, csh}` file. For example:

```
lxplus7 $ source /cvmfs/sft.cern.ch/lcg/views/LCG_94/x86_64-centos7-gcc7-dbg/setup.sh
lxplus7 $ gcc -v
...
gcc version 7.3.0 (GCC)
lxplus7 $ which root
/cvmfs/sft.cern.ch/lcg/views/LCG_94/x86_64-centos7-gcc7-dbg/bin/root
```

1.5.1 Packaging

To better address the needs of the customers, the releases are also provided in packaged form as RPMs or binary tarballs. Recently the possibility to pack the full content of a release in Docker containers has been introduced. All these artifacts are available from the EOS storage system [15], which, as described later, is an essential component of the build and continuous integration infrastructure.

RPMs. RPM format is provided for the convenience of the experiments, in particular ATLAS and LHCb, for which the LCG stack is an extension of the distribution. The RPM repository is available through the web interface of the EOS system. The path to the RPM repository is

`https://lcgpackages.web.cern.ch/lcgpackages/rpms .`

Binary tarballs. *Binary tarballs* are the initial format used for the installation of CernVM-FS; they are available for both releases and nightly builds. The path to the tarball area is

`https://lcgpackages.web.cern.ch/lcgpackages/tarFiles .`

Containers, The packaging via Docker containers has been introduced with `LCG_93` and it is still in experimental phase. It addresses the use cases where CernVM-FS is not provided/installed on the system, therefore software package have to be alternatively provided. The resulting containers images, which therefore include all the binaries of the release, are quite big, approximately 20 GB [3]. Ready-to-use containers can be found at

`https://lcgpackages.web.cern.ch/lcgpackages/docker .`

[3]To reduce the size of the generated containers to the essential, the system can include only a pre-defined subset of required packages.

1.6 Release content information

Detailed descriptions of the releases and development builds are available on GitLab. The exact content of a release or a development build, in terms of packages and available platforms, is provided by the LCGInfo web site at `http://lcginfo.cern.ch`. This site also allows to compare the content of two releases and determine the exact details about what has changed. Figure 2 shows a snapshot of the configuration differences page for releases `LCG_94` and `LCG_93c`.

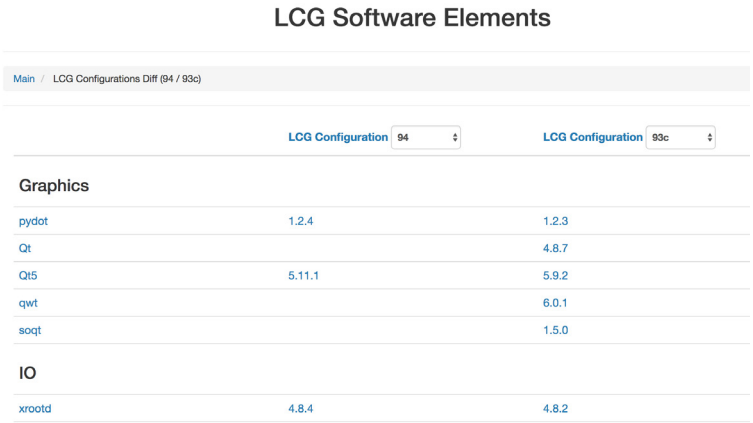

Figure 2. LCGInfo: comparison between LCG 94 and 93c content.

2 LCGCmake

The `lcgcmake` tool is based on CMake [7], a set of tools increasingly used to control the software compilation process using simple platform and compiler independent configuration files.

The main components of `lcgcmake` are:

- `LCGPackage_Add`
  A wrapper of the ExternalProject_Add function customized for LCG packages which supports: pure binary package installations, incremental builds and a central release area to avoid building already existing projects;
- The toolchains defining the detailed content of a given release, i.e. a list of package name and versions. These are CMake files listing the packages project, externals, generators being included in the release.

2.1 The `lcgcmake` GitLab repository

The `lcgcmake` infrastructure is available on CERN GitLab service

`https://gitlab.cern.ch/sft/lcgcmake`

and it is organized as a high-level CMake project. Each of the main package categories abovementioned has its own directory. The rest of common tools, modules and scripts are defined under the `cmake` directory as well as the CMake files with the toolchain definitions.

The tool can be run as a normal CMake project, with usual cmake, make, make install sequence to configure, build and install the chosen release on the local system. Alternatively, a high level interface is available through the script `lcgcmake`, providing a more user-friendly interface to the whole infrastructure [4].

3 The continuous build and integration infrastructure

To automate the whole process a Jenkins automation server is used [8]. In addition to the key tool, described above, the key elements of this system are:

Build nodes. The EP-SFT Jenkins infrastructure, which is used for all the group projects, including ROOT, Geant4 and CernVM, manages approximately 500 cores. The bulk of the build nodes are virtual machines hosted by the CERN Openstack instance, with CERN supported linux distributions flavours - currently Scientific Linux CERN 6 (`slc6`) and CERN CentOS 7 (`centos7`), the latest long-term supported Ubuntu versions (currently 16.04 and 18.04) and the supported Fedora versions (currently 27, 28 and 29). A set of Mac nodes running the supported versions of Mac Os X are also managed by Jenkins. For the purpose of the LCG stacks, the Mac and Fedora nodes are not used on regular basis.

A set of `centos7` nodes is reserved for running the build jobs inside docker containers. Pre-configured Docker containers are available through the GitLab container registry for the required flavours. Despite being rather recent, the implementation of this functionality is continuously improved and is mature enough to be used for the development builds.

EOS shared area. The EOS project used to provide the packaged releases is also used as a shared area between the build nodes. The tarballs with the sources are stored in there and the results of the builds are saved by the jobs into a dedicated path and picked up by the subsequent jobs to finalise the required objective.

Scripts. The bash and Python scripts used by Jenkins to control the required workflows are taken from a dedicated GitLab repository, `https://gitlab.cern.ch/sft/lcgjenkins` .

Dashboard. The dashboard [16], based on CDash [17], allows to monitor the results of the builds and to retrieve detailed information about the failures. The Jenkins jobs are instructed to send the information to CDash as last step before closing.

4 Summary and Future work

In this paper we have described the main aspects of the system developed and used by CERN EP-SFT to provide software stacks to a diverse community of physicist. The system is the result of many years of pragmatic development and has reached a good level of maturity and robustness, satisfying the needs of the customers: ATLAS, LCHb, SWAN, Beams and FCC.

A build infrastructure such as the one described in this paper requires continuous maintenance and research of ways for improvement. Current effort includes: improve validation and integration testing of the builds in environments as close as possible to the ones used by the experiments; keep the list of packages up-to-date; consolidate the docker setup with the goal of fully replacing the VMs; investigate the use of S3 as common shared area; reduce the publication time to CernVM-FS. And of course follow up any relevant development which could come from the HEP Software Foundation.

References

[1] CERN EP-SFT, *SoFTware Development for Experiments*, http://ep-dep-sft.web.cern.ch/

[4] The project welcome page on GitLab provides all details and options available.

[2] The ATLAS experiment, https://atlas.cern/
[3] The LHCb experiment, http://lhcb.web.cern.ch/lhcb/
[4] *The SWAN Service*, https://swan.web.cern.ch
[5] *Future Circular Collider Studies*, https://fcc.web.cern.ch
[6] CERN Beams Department, https://beams.web.cern.ch/
[7] Kitware, Inc., *CMake*, https://cmake.org/
[8] *The Jenkins System*, https://jenkins.io
[9] A Valassi, *HEPOSLibs*, https://gitlab.cern.ch/linuxsupport/rpms/HEP_OSlibs
[10] *The ROOT Data Analysis Framework*, https://root.cern
[11] *The Geant4 Simulation Kit*, https://geant4.web.cern.ch/
[12] *The Persistency Framework*, https://twiki.cern.ch/twiki/bin/view/Persistency
[13] B Hegner, *HSF Platform Naming Conventions - A Proposal*, HSF-TN-2018-01
[14] *The CernVM File System*, http://cernvm.cern.ch/portal/filesystem
[15] *The EOS system*, http://eos.cern.ch/
[16] See LCGSoft at http://cdash.cern.ch
[17] Kitware, Inc., *CDash*, https://www.cdash.org/