Journal of Physics: Conference Series **513** (2014) 052009 doi:10.1088/1742-6596/513/5/052009

Rise of the Build Infrastructure

Giulio Eulisse
Fermi National Accelerator Laboratory, USA
E-mail: giulio.eulisse@cern.ch

Shahzad Muzaffar
Fermi National Accelerator Laboratory, USA

David Abdurachmanov
Vilnius University, Lithuania

David Mendez
Universidad de los Andes, Colombia

Abstract.
CMS Offline Software, CMSSW, is an extremely large software project, with roughly 3 millions lines of code, two hundreds of active developers and two to three active development branches. Given the scale of the problem, both from a technical and a human point of view, being able to keep on track such a large project, bug free, and to deliver builds for different architectures is a challenge in itself. Moreover the challenges posed by the future migration of CMSSW to multithreading also require adapting and improving our QA tools. We present the work done in the last two years in our build and integration infrastructure, particularly in the form of improvements to our build tools, in the simplification and extensibility of our build infrastructure and the new features added to our QA and profiling tools. Finally we present our plans for the future directions for code management and how this reflects on our workflows and the underlying software infrastructure.

1. Motivations for an improved build infrastructure

CMS Offline Software (CMSSW) build infrastructure and associated tools have been widely presented by CMS in a variety of occasions ([1], [2], [3], [4]). They allow a rather large and non homogeneous community of a few hundreds physicists and software engineers to still deliver and deploy a coherent software release which is used for all the needs of CMS Offline operations. This is made possible by centralized bi-daily integration builds for all the open release series and a conspicuous amount of unit tests, integration tests, and QA efforts ([5]).

They key components of such an infrastructure are:

- A software repository using a version control system (CVS until recently).
- A set of in house tools, which go under the name of PKGTOOLS, to describe build and deployment recipes in a reproducible manner.
- A build tool, SCRAM, which is finely tuned to handle a large amounts of directories ("packages" in CMS-speak), which reduce the conflicts between developers, at the cost of increasing the complexity of the dependency graph between various parts of the system.
- An in house web application, backed by an Oracle DB, responsible for keeping track of the various versions of the above mentioned packages and to organize them into a coherent release.

Despite the design flaws of some of the tools (most notably CVS) and the ad-hoc nature of the whole system, this has worked out quite well for us since for the last years, for the whole preparation and duration of the so called Run 1 of the CMS experiment.

However the design of the whole system dates back to as long as fifteen years ago, and its roots are even older than that. The additional requirements and external constraints which were added over the years have forced us to stretch the original design and rethink only minor parts of it given the impossibility of major changes during data taking. In particular, there are three major problems for the old design:

- The reliance on CVS and the CVS service provided by CERN.
- The maintenance of a complex in house web application (CMS Tag Collector) to drive the integration needs.
- The binding between a build machine and a given release series for a given architecture.

The first point became problematic when CERN/IT announced the desire to stop the CVS hosting service, the second point became critical due to the fact that the specialized manpower required to develop and maintain such a system had to be moved or shared with other tasks and finally the third point started to create problems when the number of release series and architectures to support increased due to overlapping needs of data taking, analysis and R&D for the next Run of the experiment. CMS therefore decided to use the Long Shutdown of the LHC as an opportunity to look back at what we did and redesign the parts of the build infrastructure which proved ineffective or non-efficient. As a guiding strategy for our decision making process we have decided to be as aggressive as possible in selecting technologies and solutions which minimized the need for in-house customizations or maintenance man-power.

2. Dynamic build infrastructure

When we talk about CMSSW release builds we actually mean the following workflow:

- Actual build of the release (an average of 2:30h, depending on the release, the compiler used, and of course on the machine)
- Unit testing (∼0:30h)
- Integration tests and physics validation (∼6h)
- Quality Assurance tests (∼3h)

This workflow is repeated between 20 and 30 times per day, delivering twice per day integration builds for a variety of combinations of platforms, release series, with occasional special branches used to facilitate the integration of larger feature sets or to tackle large scale migration to new version of some core tool like ROOT6, a new version of Geant4. In the original design we assigned each release, architecture pair to a specific machine triggering the build via a cron job. This approach has several limitations though:

- It has a single point of failure in the build machine.
- It wastes resources.
- It is very basic and required to develop many components for coordination of builds and to present result, logs and analytics about a given build.

To improve on these issues, we decided to cut the binding between a given machine and a the associated release. This meant that build machines really became expendable goods, which can do each other work (in the limit imposed by the compatibility with a given architecture) and can be quickly replaced if there is need to do so. We can use the spare time of newer, faster boxes to take care of some of the tasks which would have been bound to less performant ones or, in case we have resource shortage for any reason, it still allows us to get the job done by lifting the requirements that all builds need to start at the same time. Finally as a by-product of this decision, thanks to appropriate scheduling, it became possible to use resources assigned to developers by policy when they were are unused. We considered a number of open source alternatives when looking for a build scheduling system, including buildbot, CDash, Jenkins and even writing our own. buildbot and CDash while they are both excellent tools, proved to be either too limited in scope or to require rather large amounts of additional code to made behave as required. In the end the choice was between Jenkins and writing our own. While we tinkered with the idea of writing our own and had an initial version also used in production for a while, we decided that we did not have the resources to develop and maintain it so in the end we selected Jenkins [6], a JAVA based continuos integration system widely used both in the open-source world and in the industry. While it was born with the typical JAVA build workflow in mind, thanks to its extremely developed plugin system, Jenkins provides enough flexibility to handle ad-hoc build workflows like the CMSSW one. We now use Jenkins for a wide spectrum of tasks ranging from simple backup jobs to more complex build pipelines. In all cases we found of particular interest the various analytics and trend information provided which were missing in the previous system and which simplify a lot debugging of infrastructural problems, without having to parse hundreds of mail reports which was the case of the previous system.

3. Adoption of git as basis for a new development model

The move to a dynamic build infrastructure has allowed us to rethink in general the way we look at release integration since we now have the flexibility to use idle resources for integration tasks which were previously done by hand. In particular we wanted to embrace for our release workflow the so called "Continuous Integration" approach, where a reduced test suite to be run in a semi automatic way for proposed updates to the release, before they actually enter in the IB, and possibly before the relative code gets integrated in the official release branch. In such a development we have:

- A release branch, which we try to make sure we can always cut a release from.
- Many so called topic branches, where developers prepare their new additions or bug-fixes which they request for inclusion in the release branch when ready. Such branches get merged back to the release only once they are reviewed by the affected software area coordinators and pass a limited (but meaningful) test suite.

The main advantage of this approach is that it reduces the clashes between various developers by exposing them only to each other finished and approved development.

This new workflow has obviously deep interactions with the version control systems (VCS) used to manage the source code. While most of the tools usually used for this provide some branching mechanism, including the old CVS, it was obvious that the one which maps better and has most momentum behind it is git [7] which was designed with this exact workflow in mind, given that's the typical approach.

The migration to git has been done using widely used open source tool called cvs2git [8]. As a strategic choice we decided to have the whole per file CVS time ordered history for all the files which entered in any of our past CMSSW releases. On top of this linear, per-file, commit history, we forked at the correct point in time one commit branches which would have all the changes which transform the CVS HEAD of given point in time to the actual content of the release tag. We migrated this way the contents of all the production release tags we had. This approach has the advantage of avoiding duplicating commits in the per file history making it closely resemble the CVS history when looked at on a per file basis, however it does introduce complications due to the fact that two logically subsequent releases (e.g. CMSSW_6_2_0 and CMSSW_6_2_1) are not one derived one from the other in the git history. They are parallel branches, consisting of one single commit on top the branch which contains the time ordered history. An alternative, and possibly better under certain aspects, approach would have been to forget about the time ordered CVS history and migrate the release tags keeping into account the logical release order. The migration of the 7GB CMSSW CVS repository took 14 hours, a peak of 23GB of RAM and 98GB of temporary disk space. The result was a git repository of 350MB, which shows how much more compact git repositories are when compared to CVS ones due to the fact data is actually compressed. The parts in the repository which were not included in any release, most notably the personal user code of single users or analysis group, were migrated independently, with migration choices which varied on a case by case basis.

For the choice on where to host the authoritative repository of CMSSW, we decided to abandon the CERN-centric approach and look for alternatives from industry. This was driven both by the fact that the CERN git service was not even yet introduced when this investigation started and there was no real experience on how it would have performed and by the desire to design the new system around site-agnostic assumptions, given the past experience with the limitations of a laboratory-centric design. There were of course many hosting options possible, including highly regarded providers like BitBucket, Gitorious, Sourceforge, or even Google own Google Code service but in the end we selected Github, basing our choice on the features provided, availability of a public API to the service, the number of users and repositories already hosted, and general reputation. One of the driving motivation for the choice was also that the Linux kernel, i.e. the other main contribution to humanity of the git creator, is hosted on Github. If Linus Torvalds is happy about it, chances are CMS will be more than happy as well. This abundance of choices for git service providers and git own distributed design nature also convinced us that the risk involved in picking up an external provider was minimal and that our hosting model would have survived even in the event of catastrophic problems with Github.

Since in the meanwhile CERN started to provide a git service as well, we decided to have a comparison between the two. You can find the results of our performance measurements in the next section, however the main result is that once we started to use a AFS cache and the obvious locality advantage of CERN/IT service was leveled out, Github proved to be as performant, if not better. The choice was therefore driven solely by integration and collaboration features provided, like forked repositories and pull requests, which at this time are Github only.

We therefore decided to select Github as the hosting service and to map our Development Model on top of its key features, which allowed us to minimize the amount of custom code required and in particular to get rid of the CMS Tag Collector component of the old CVS based workflow. The CERN/IT provided service is kept only as a backup of the Github repository.

The new authoritative CMSSW repository, `cms-sw/cmssw` [9], is a standard, public, Github repository set up so that only release managers can push changes to it. CMS developers fork such a repository into a personal one and develop in it. Once a new feature or bug fix is ready to be included in the official release branch, they open a so called pull request to the official repository. A custom CMS chat bot drives the integration process, by notifying coordinators for the affected software areas. A number of integration tests are run on the proposed update,

Table 1. Release download performance.

	Time (s)
Github full clone, no reference repository	140
CERN full clone, no reference repository	110
Github full clone, with reference repository	23
CERN full clone, with reference repository	24
Github full clone, with reference repository, no checkout	13
CERN full clone, with reference repository, no checkout	17
CVS checkout of a tagged release	1159

via a configurable Jenkins job which is triggered by the release manager. When all the involved parties give their sign-off, via a +1 comment in the message board associated to a given Pull Request, the bot notifies the release manager which then can merge the changes or request further updates to the fix. In the latter case the bot will notice an update to the pull request and reset the signatures so that no change goes in unreviewed.

While we still have limited experience with git and Github, having migrated our repository at the beginning of the summer, we have gathered already quite a lot of experience. The most positive ones regard the Github service itself which has proven extremely stable for us, and despite the fact Github is from time to time preferred target of "denial of service" attacks the distributed nature worked as expected in mitigating the need to access the authoritative repository since most of the development tasks can work in a disconnected manner using the local one. While we have no ways to really verify the 99.83% uptime claimed by Github itself for the last month (and similar if not higher values for the preceding months), not a single delay in our release building process in over 4 months of usage can be blamed to Github being down. Moreover many of the user interface features provided by Github have been widely appreciated by some of the developers most hostile to the transition. In particular the whole "Social network like" system to comment on code and bugs has been widely appreciated and is considered a killer feature.

4. Untitled

5. Performance comparisons

When evaluating git and Github, we also did tests to evaluate their performance compared to CVS and CERN/IT provided git service. We found that using the `--reference` option of git, which allows us to use a local reference copy for git db object store, therefore avoiding the remote download of most of the repository, brought the initial overhead down to an handful of seconds and completely eliminated the obvious bandwidth advantage CERN has vs the vanilla Github. This feature also allows us to keep the overhead of a work area under control since object that are found in the reference copy are not stored in the local object store.

Another feature which helped us to keep the size of the work area under control is the sparse checkout, which allowed us to mimic the behavior of the of the old `addpkg` wrapper used for the CVS case. Moreover we notices that while CVS has still an advantage when checking out small packages, however such an advantage disappears when a larger number of packages is checked out.

All the measurements were done on a physical build machine hosted by CERN/IT with a standard SATA disk. Finally during our operations in the last months, we noticed CERN/IT provided service suffers of young age problems. We rely on it only for daily backups of the

Table 2. Partial checkout performance and overhead.

	Time (s)	Size (MB)
Github checkout FWCore/Framework package	13	13
Github checkout FWCore subsystem	14	34
CVS checkout of FWCore/Framework	1	5
CVS checkout of FWCore subsytem	13	38

Github repository yet we observe a 10% failure rate when pushing to it on a daily basis.

6. Conclusions and further work

We have been able to migrate a large heterogeneous collaboration to git and Github in less than one year, without major disruption of CMS release integration process. Moreover we have set the basis for a much improved build infrastructure, based on Jenkins, and most important a more streamlined development process which will hopefully improve the quality of our software by pre-emptive peer-reviews of new features and bug-fixes. While migrating a large set of people to a new VCS is not an easy task, we are confident that the learning curve for the new tool will flatten out. We cannot help not noticing that younger contributors which have been exposed less to the old way of working and which use git for their own personal projects have of course less problems than CVS gurus. For what concerns further work, one addition we would like to deploy is the usage of cloud technologies within Jenkins, in particular the OpenStack based CERN Cloud [10] to further scale the build infrastructure and adapt it to varying workloads.

7. Acknowledgements

We would like to thank CMS Offline Management, Elizabeth Sexton Kennedy, Fabio Cossutti and Peter Elmer for the support given throughout the project. This work has been partially supported by the Department of Energy and the National Science Foundation of the United States of America.

References

[1] Eulisse G. et al, CMS packaging system or: how I learned stop worrying and love RPM spec files, Computing in High-Energy and Nuclear Physics (CHEP), Victoria, 2007.

[2] Pfeiffer, A. et al, CMS software Infrastructure Tools. Published in J. Phys. Conf. Ser. 219 (2010) 042047.

[3] Lange, D. et al, Software Integration and Development Tools in CMS, Computing in High-Energy and Nuclear Physics (CHEP), Prague, 2009.

[4] Muzaffar, S. et al, Optimization of the CMS software build and distribution system, Computing in High-Energy and Nuclear Physics (CHEP), Prague, 2009. Published in J.Phys. Conf. Ser. 219 (2010) 042047.

[5] Elizabeth Sexton-Kennedy (for the CMS collaboration), Release strategies: The CMS approach for development and quality assurance. Published in J. Phys. Conf. Ser. 331 (2011) 042023.

[6] http://jenkins-ci.org

[7] http://git-scm.org

[8] http://cvs2svn.tigris.org/cvs2git.html

[9] http://github.com/cms-sw/cmssw

[10] Andrade, P. et al, Review of CERN Data Centre Infrastructure, J. Phys.: Conf. Ser. 396 042002