EPJ Web of Conferences **214**, 05001 (2019) https://doi.org/10.1051/epjconf/201921405001
CHEP 2018

LHCb Continuous Integration and Deployment system: a message based approach

Stefan-Gabriel Chitic[1,*], *Ben* Couturier[1,**], *Marco* Clemencic[1,***], and *Joel* Closier[1,****] on behalf of the LHCb collaboration

[1]CERN, European Organization for Nuclear Research, Geneva, Switzerland

Abstract. A continuous integration system is crucial to maintain the quality of the 6 millions lines of C++ and Python source code of the LHCb software in order to ensure consistent builds of the software as well as to run the unit and integration tests. Jenkins automation server is used for this purpose. It builds and tests around 100 configurations and produces in the order of 1500 built artifacts per day which are installed on the CVMFS file system or potentially on the developers' machines.

Faced with a large and growing number of configurations built every day, and in order to ease inter-operation between the continuous integration system and the developers, we decided to put in place a flexible messaging system. As soon as the built artifacts have been produced, the distributed system allows their deployment based on the priority of the configurations.

We will describe the architecture of the new system, which is based on RabbitMQ messaging system (and the pika Python client library), and uses priority queues to start the LHCb software integration tests and to drive the installation of the nightly builds on the CVMFS file system. We will also show how the introduction of an event based system can help with the communication of results to developers.

1 Introduction

Starting from 2019, the LHCb experiment will start the detector upgrade for the next stage of data taking [2] of the Large Hadron Collider (LHC). The existing computing infrastructure and software need to be updated [3]. Having a large code base with a large number of developers involved in its evolution, LHCb software needs a modern infrastructure capable of handling the correct compilation, testing and deployment to ensure the quality of executable software products, called artifacts. This can be assured by testing the software improvements using the Jenkins [4] continuous integration system to drive the nightly builds. It needs to build and test around 100 configurations (i.e. combinations of multiple projects) that form around 1500 nightly built artifacts which need to be deployed on a centralized location available for performance tests run by the users.

*e-mail: stefan-gabriel.chitic@cern.ch
**e-mail: ben.couturier@cern.ch
***e-mail: marco.clemencic@cern.ch
****e-mail: joel.closier@cern.ch

In the past, AFS [5] was initially chosen as the centralized location to install the nightly builds. However, as of today, CERN support for AFS is phasing out [6]. CVMFS [7] has been chosen to replace AFS. CVMFS is a read-only file system that provides a scalable, reliable and low-maintenance software distribution service. The CVMFS deployment is laborious and slow because each installation needs to be done on a special node with writing privileges called *Stratum-0*. Furthermore, each file modification needs to be included in a *transaction*. Each transaction needs to be serialized since no parallel transactions can co-exist.

In order to cope with the growing number of configurations built every day and to cope with the limitations of the deployment environment, we introduce a flexible messaging system that allows us to test and deploy a software configuration based on its priority as soon as it is built. This paper describes how such a system based on the RabbitMQ [8] messaging broker is used to deploy these configurations on CMVFS. The paper is structured as follows: Section 2 presents the global architecture of the continuous integration system and argues the usage of RabbitMQ messaging system and Section 3 focuses on the deployment systems on CVMFS and how the nightly builds are installed. Section 4 presents the future works on this system and how new use-cases can be developed from the existing RabbitMQ infrastructure. Section 5 concludes the paper.

2 Distributed continuous integration system

In order to allow the new development to be tested and used in LHCb, we have proposed a complete integration system that starts from users latest commits. Every night, this code gets built, tested and deployed on CVMFS, from where users can use their nightly builds in integration tests. This section focuses on the architecture and the technologies used to provide users with such distributed system.

2.1 Architecture

The LHCb continuous integration system is a distributed, loosely coupled and complex system that relies on various technologies that include GitLab [15], Jenkins [4], RabbitMQ [8], and CouchDB [16] servers.

As shown in Fig.1, the starting point of a nightly build is represented by the new development committed by the users into the git repository (GitLab). Each night a Jenkins scheduler triggers the build of every configuration, called slot. Each build has a unique incremental build id for a given slot. Inside each slot, multiple projects are combined in order to form a software stack. Each project can be built for multiple combinations of operating systems, compilers and optimization flags, called platforms.

The build sequence starts by checking out the git repository for each project in each slot and triggers the compilation of the artifacts. Each project is built in separated environments running inside Docker [17] containers. The results of the builds are copied to a shared location (used to be able to exchange the build products between the distributed build nodes and the deployment node in the setup) in the EOS [18] system. Once a build is completed several events happen:

- a new Jenkins job is triggered that test the artifacts against predefined tests;
- the output of the build system is persisted inside CouchDB. This will allow for reports to be generated later in the system;
- a message is pushed to the RabbitMQ broker to notify the other components of the system that an artifact was successfully built.

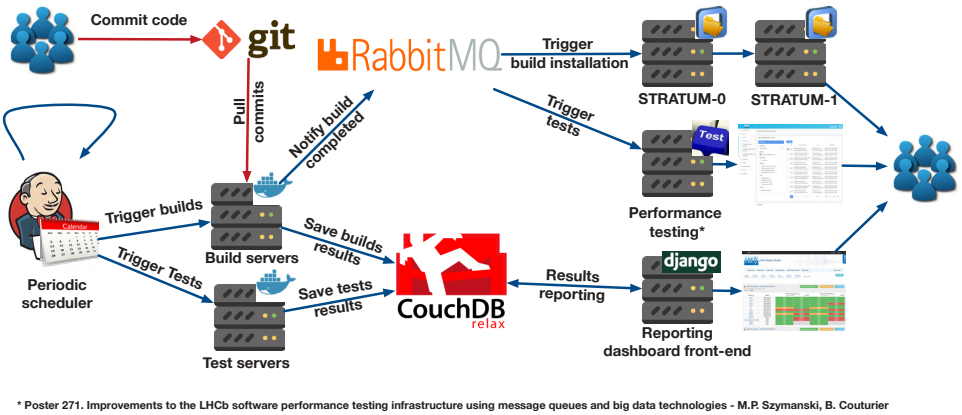

Figure 1. General architecture of LHCb continuous integration and deployment system

The results of the tests are also synced to CouchDB. This document-based database allows us to store information about the build and test results for each combination of slot-build id-project-platform. As shown in Fig.1, this information is used by the reporting front-end to display reports. This front-end is represented by a Django [19] web-application that generates reports in a cross-device, cross-platform user friendly interface.

The RabbitMQ brokering receives the notification of a successfully built artifact, and routes the message to two components: (i) The performance testing (PR) described in [20], (ii) The CVMFS deployment system that will trigger the installation of the artifact on a special node with writing privileges, called *Stratum-0*. This installation will later be synchronized to the nodes visible by users in reading mode, called *Stratum-1*.

2.2 Why RabbitMQ as message broker?

The choice of RabbitMQ as a message broker was decided based on the features offered by a variety of messaging systems (e.g ActiveMQ [9], ZeroMQ [10], Kafka [11] etc), their compatibility with different programming languages and our desire to have a controlled instance of the system.

RabbitMQ provides support for multiple messaging protocols like *AMQP [12]*, *MQTT [13]*, etc. In "Advanced Message Queuing Protocol" (*AMQP*), in order to allow interoperable messaging middleware, both the network protocol and the semantics of the broker need to be sufficiently specified. Therefore, *AMQP* defines two major components: (i) A network wirelevel protocol that provides the clients communication to the broker and interacts with the *AMQP* model by providing data serialization (called framing) and aliveness checks via heartbeats, (ii) AMQP Model that consists of a set of components defining routing and storing schemes for the messages as well as a set of rules to wire those components together. The *AMQP* model is exposed in the client APIs while the network level is hidden from the users in client libraries. In our solution we have chosen to use AMQP beacuse this decopeling between the exchanges and queues allows for a better and transparent management of the message delivery.

For performance reasons, the user can choose between reliability (including persistence, delivery acknowledgements, publisher confirms) and high availability. Several RabbitMQ servers can be clustered together, forming a single logical broker. In the case of more loosely

and unreliably connected servers, RabbitMQ offers a federation model. Another key feature is the flexible routing since it provides message routing through exchanges before arriving at queues.

Furthermore, the servers comes with a complete management user interface that allows one to monitor and control the broker. It also includes a plugin system and a large number of community supported libraries for different programming languages (e.g. pika [14] for Python).

2.3 RabbitMQ usage in LHCb continuous integration system

All the software components takes advantage of the ability of the message broker to prioritize messages that will be explained in the next section. All the software components involved in communicating with RabbitMQ are easily developed, managed and used in Python using the pika package ("a pure-Python implementation of the AMQP protocol that tries to stay fairly independent of the underlying network support library". [14]).

As already shown in Fig.1, RabbitMQ plays a vital role in the LHCb continuous integration architecture. It is mainly used as a message bus between different components of the system which allow the decoupling of message producers from consumers on different nodes at different stages in the system. This makes the system highly distributed, fault tolerant and loosely coupled. Furthermore, allowing for persistent queues renders the system capable of recovery even if the message broker fails.

3 Deployment system

After an artifact is built in the continuous integration system, the continuous deployment system is informed via a RabbitMQ message. Its role is to install the nightly products on the CVMFS shared repository. This section focuses on the components that are involved in this system as well as why they are needed.

3.1 Priority policy

In order to better understand the deployment system, one should notice how a CVMFS file modification takes place. Each write operation on CVMFS happens only on a special node, *Stratum-0*. Here all the operations are encapsulated in transactions which guarantees the atomicity of writing when publishing. Multiple transactions cannot exist in parallel. They are serialized, meaning that a transaction needs to be published before the system can trigger a new transaction. The publishing time of a transaction is far greater than the file operations themselves, resulting in a bottle neck for our deployment system.

Since the continuous integration system is building the artifacts in parallel on multiple nodes, the deployment system will face a "burst" effect from the build servers. A priority policy is needed in order to allow more important artifacts to be installed first. This policy can be changed based on the needs of users. This means that the ordering should be transparent for the deployment components on the CVMFS *Stratum-0*.

In previous versions of the deployment system we used to pull all the available artifacts (without taking into account any priority), install and publish everything in one transaction. This took a long time to finish and an even longer time for the installations to be propagated to the users. The later iterations decoupled the deployment in smaller transactions because it resulted in installations being propagated faster.

In the current version of the system we have opted for prioritized and better distribution of the installations. First of all, the most important platforms of all projects in the slots are

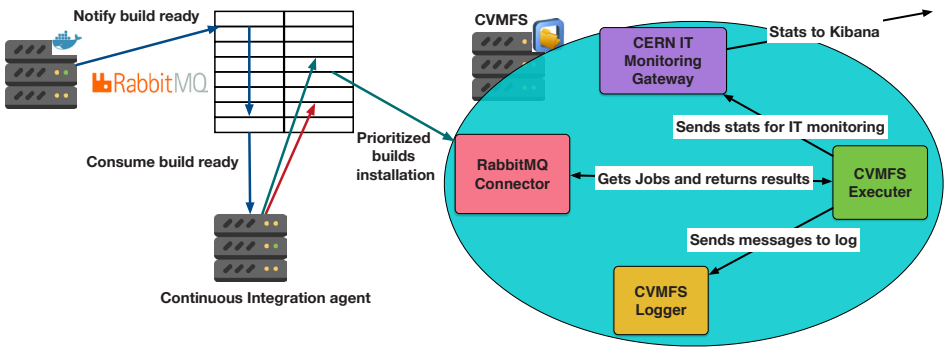

Figure 2. Architecture of LHCb continuous deployment components

installed with a higher priority. Then, all the projects with all the platforms inside each slot are installed using a slot based on priority from a configuration file.

3.2 Architecture

The deployment system is composed of three major distributed components on different nodes: the continuous integration agent that prioritizes the installations, the CVMFS scheduler, and the executor.

As shown in Fig. 2, the continuous integration agent is the first component that gets the push notifications via RabbitMQ from the continuous integration system. Its main role is to apply the priority policy based on the slot and platform of the artifact. Then, it pushes the updated message with the right priority attached back to the RabbitMQ broker. This mechanism allows the CVFMS scheduler, that resides on *Stratum-0*, to consume the messages based on their priority instead of a first-come-first-consumed policy. Using decoupled components, the order of the installations can be changed during a day installation through the continuous integration agent. It also allows for the possibility of injecting/removing installations manually.

The CVMFS Scheduler, on *Stratum-0*, gets the messages and ensures that a transaction can be started. When a new artifact needs to be installed, it triggers the CVMFS Executor (as shown in Fig.2) which takes care of starting a new transaction, downloading the artifact from EOS and installing the files in the right path. After the installation process is completed, it triggers the publishing of the current transaction. If an error occurs during the installation process, the transaction is aborted. The CVMFS executor allows for a better management of errors using a separate queue in RabbitMQ.

The executor sends statistics on the time taken to install each pair of slot-project-platform to CERN Monitoring infrastructure [21] and also logs locally all the relevant information about each artifact deployment using the CVMFS Logger.

4 Future work

The current system is stable in terms of reliability. For the next version of the deployment system, we want to improve the deployment time of the system, i.e. to reduce the time from an artifact is produced until it is available for the user on CVMFS. Besides providing our

users with a better service, this will allow us to be able to scale with an increasing number of configurations (slots) related to the software development needed for the upgrade. Moreover, we have started looking into clustering multiple instance of RabbitMQ servers to avoid the single point of failure.

Since the messaging infrastructure is already in place, we can take advantage of the new message bus by notifying other distributed components of the continuous integration system or even by informing users about the status of the system. There is already a communication channel with the users through Mattermost [22], where an automatic update is pushed each time an artifact is deployed. Finally, an improvement for the monitoring and error management is needed in order to reduce the operations tasks.

5 Summary

In this paper we have presented the architecture, the technologies and the software developed to improve the LHCb continuous integration and deployment system.

Providing an end-to-end continuous integration and deployment system allows LHCb software developers to have their development built, tested and deployed each night in order for them to use their work the next day. Using a messaging bus, RabbitMQ, allows the system to decouple components on different nodes making the distributed system more fault tolerant and less error prone. The new developed system is composed of software that sends build ready notifications, prioritizes installations based on specific rules, transfers and deploys the artifacts on CVMFS based on pushed messages from RabbitMQ. The new deployment system allows for flexible and automatic installations instead of manual installations which are both error prone, laborious and slow. Even if the system described in this paper is complex and distributed, it is developed and monitored with Python which allows us to easily prototype future evolution. Having the infrastructure in place opens the door to opportunities using the new messaging bus.

References

[1] M Clemencic and B Couturier "A New Nightly Build system for LHCb" J.Phys.Conf.Ser, Volume 513, Track 5, p 052007 (2014)

[2] I. Bediaga *et al.* (LHCb Collaboration), "Framework TDR for the LHCb Upgrade : Technical Design Report" (CERN-LHCC-2012-007, LHCb-TDR-12, 2012)

[3] The LHCb Collaboration, CERN, "Upgrade Software and Computing," (CERN-LHCC-2018-007. LHCB-TDR-017, 2017)

[4] Jenkins (software), https://jenkins.io/ (accessed 2018-10-25)

[5] HOWARD, John H., et al. "An overview of the andrew file system." (Carnegie Mellon University, Information Technology Center, 1988).

[6] J. Iven, M. Lamanna and A. Pace, "CERN's AFS replacement project," J. Phys. Conf. Ser. 898 no.6, 062040 (2017)

[7] P. Buncic, C. Aguado Sanchez, J. Blomer, L. Franco, A. Harutyunian, P. Mato and Y. Yao, "CernVM: A virtual software appliance for LHC applications," J. Phys. Conf. Ser. 219 (2010)

[8] RabbitMQ (software), https://www.rabbitmq.com/ (accessed 2018-10-25)

[9] ActiveMQ (software),http://activemq.apache.org/ (accessed 2018-10-25)

[10] ZeroMQ (software),http://zeromq.org/ (accessed 2018-10-25)

[11] Kafka (software), https://kafka.apache.org/ (accessed 2018-10-25)

[12] AMQP (software), https://www.amqp.org/ (accessed 2018-10-25)
[13] MQTT (software), http://mqtt.org/ (accessed 2018-10-25)
[14] Pika (software), https://pika.readthedocs.io/en/stable/ (accessed 2018-10-25)
[15] Gitlab (software),https://about.gitlab.com/ (accessed 2018-10-25)
[16] CouchDB (software), http://couchdb.apache.org/ (accessed 2018-10-25)
[17] Docker (software), https://www.docker.com/ (accessed 2018-10-25)
[18] EOS (software), http://information-technology.web.cern.ch/services/eos-service/ (accessed 2018-10-25)
[19] Django (software), https://www.djangoproject.com/ (accessed 2018-10-25)
[20] Szymański, Maciej., et al. "Improvements to the LHCb software performance testing infrastructure using message queues and big data technologies" in this proceedings (2018).
[21] Monit (software),http://monit.web.cern.ch/monit/ (accessed 2018-10-25)
[22] Mattermost (software),https://mattermost.com/ (accessed 2018-10-25)