Prometheus-Powered Insight: Monitoring Koji's performance

Marta Vila Fernandes[1,*]

[1]IT Department, CERN - 1211 Geneva 23 - Switzerland

Abstract. Efficient, ideally fully automated, software package building is essential in the computing supply chain of the CERN experiments. With Koji, a very popular software building system used in the upstream Enterprise Linux communities, CERN IT provides a service to build software and images for the Linux OSes we support. Due to the criticality of the service and the limitations in Koji's built-in monitoring, the CERN Linux team implemented new functionality to allow integration with Prometheus, an open-source monitoring system and time-series database. This contribution will give an overview of Koji and its integration with Prometheus and Grafana, explain the challenges we tackled during the development of the integration, and how we're benefiting from these new metrics to improve the quality of the service.

1 Introduction

The CERN Linux team is responsible for defining the Linux strategy for the organization, providing support for package and image building, as well as for the deployment and operation of the supported Linux distributions on servers and managed desktops. The Linux operating system is heavily used at CERN for several use cases, including CERN experiments, accelerators, and IT. Figure 1 shows the number of Linux hosts per CERN department/group. The majority of the Linux machines are managed by IT services (for instance Lxplus and batch services), but also the Beams and Experimental Physics departments are large consumers of the Linux services. Of the approximately 13,800 (physical and virtual machines) in the Data Center, about 13,200 hosts are running Linux. There are also approximately 10,500 hosts running Linux at CERN outside the Data Center. However, this number can be underestimated because some experiments have their own mirrors of our content in the CERN technical network, that has been designed to be inaccessible from outside CERN, and their machines don't access our services directly.

These numbers show the high importance of having an automated, consistent and reliable software package building system, such as Koji, to provide a service to build software and images for the Linux operating systems supported at CERN, that can be used across the all CERN community, institutions, universities, and other particle accelerators.

2 Koji build system at CERN

Koji is an open source tool, written in Python, used and developed primarily by Red Hat employees. It is a system for building and tracking binary RPMs, Cloud and Docker images.

*e-mail: marta.vila.fernandes@cern.ch

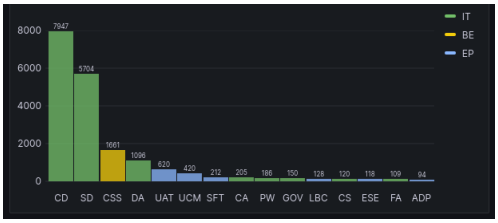

Figure 1. Number of host by CERN department/groups

Koji enables users to create tags, that represent internal repository names. The user submits a source RPM or a git repository, and Koji builds a binary RPM associated with a tag. Koji supports building for many architectures, such as `x86_64`, `aarch64` and `i386`, which are the ones being used at CERN.

CERN Koji has around 2700 tags, approximately 8200 packages and thousands of tasks, including builds.

The goal of this work is to show how Prometheus can be used to retrieve Koji's metrics that helps to monitor its performance and to guarantee the good delivery of the service.

2.1 Koji components

Koji is comprised of several components:

- **koji-hub** is the center of all Koji operations. It is the only component that has direct access to the PostgreSQL database and is one of the two components that have write access to the shared file system.
- **kojid** is the build daemon that runs on each of the build machines. Its primary responsibility is polling for incoming build jobs and handling them accordingly.
- **kojira** is a daemon that handles buildroot repos. It checks if any builds were added to buildroot or build tag configuration has changed. If so, it triggers a task to update the buildroot.
- **koji-web** is a web frontend to Koji.
- **koji-client** is a CLI written in Python that allows users and admins to interact with Koji.

At CERN, the service is composed of two environments, test and production. It has managed AlmaLinux and RHEL machines, using Puppet for configuration.

The production service has two hub machines, three web machines, and ten machines with kojid installed, called builder machines. There are six builder machines for `x86_64` and `i386`, and four builder machines for `aarch64`.

2.2 Koji client configuration

The Koji client is available as part of the Koji package. The upstream package is rebuilt at CERN every time there is a new version, because that way we can include patches needed to be supported at CERN, and CERN customisations. It is firstly deployed in the testing environment and kept during one week before deploying to production, to ensure the new version didn't introduce any regressions.

To use the CERN Koji service, each user needs to request access to the Linux Team, which has an automated process to grant build permissions. The synchronization script is scheduled to run every day within a container in Nomad, which is a workload orchestrator used for automating tasks. It is important to control service's users, what tags they are responsible for, and Koji's ACLs to avoid security vulnerabilities.

The users are encouraged to use the centralized interactive logon services, where the Koji client is installed and centrally managed.

Koji website is internal accessible in koji.cern.ch, and it uses Kerberos for authentication.

2.3 Koji concepts: Tags, Packages and Hosts

Koji organizes packages using tags, and all CERN tags in Koji have a corresponding internal repository on http://linuxsoft.cern.ch/internal/repos.

There are three different concepts about packages in Koji: the package itself, a build, and the RPM. The package is the name of a source RPM. A build of a package includes all the architectures and subpackages. An RPM created by a build operation has a specific architecture and subpackage of a build.

For instance, mytag9al−testing tag has the package, myrpm.

- **Tag:** mytag9al−testing
- **Package:** myrpm
- **Build:** myrpm−1.1−29.el9
- **RPM:** myrpm−1.1−29.el9.x86_64.rpm

An RPM is composed of a name, version, and release (which contains a disttag), the NVR. It is unique in Koji. In the example above, the RPM name is `myrpm`, version `1.1`, release `29.el9`, distag `el9`, and arch `x86_64`.

An RPM is built with `rpmbuild` command and to generate a clean buildroot, Koji uses Mock. To release a new RPM, the release number associated with it needs to be increased in the `.spec` file.

In Koji web, there is a Hosts tab, where information about the Koji builders can be found. This information includes the name of the host; the architectures that can be built by each host; The channels are the general Koji methods associated to each host, it comprises of `createrepo`, `default`, `image`, and others. For instance, some hosts are used to build images, or build RPMs, or both. Each channel has different methods, like `rebuildSRPM` or `newRepo`. There is also the concept of `Enabled` and `Ready`. A node can accept Koji requests only if it is enabled. A node can be enabled but not ready, and sometimes it means that the `kojid` process is not running, or the host is overloaded. The host can be enabled or disabled via Koji web button. Tasks have different weights, a decimal number represents them. The capacity is the total weight of the tasks per node, and the load is the total weight of all the tasks running at the moment.

2.4 Build operations

Koji can be used to request package builds and get information about the buildsystem. At CERN, an RPM can be built from a `src.rpm` or from a version control system, like git.

Koji uses Mock to build RPM packages for specific architectures and ensure that they build correctly. Mock creates chroots and builds packages in them. A chroot ("change root") is an isolated root directory that allows you to test safely without compromising the real root

directory. Mock's task is to reliably populate a chroot and attempt to build a package in that chroot.

All the build operations can be executed with `-scratch` to test if the package will build correctly. The scratch build is useful to be able to build a package against the buildroot, but without actually finalizing the NVR, meaning that you can build multiple scratch builds with the same NVR.

Every package needs to be added to a tag before being built.

There are useful commands to check that the RPM package was built, and it is available in the <tag_name> repository:

- Get the packages that belong to a tag: `koji list-pkgs -tag <tag_name>`
- Get the latest build of a tag: `koji latest-build -all <tag_name>`
- Get all the builds given a package: `koji list-builds -package <packake_name>`

2.5 Koji built-in monitoring

Koji web has a tab called Reports. The reports available are: number of packages owned by each user, number of builds submitted by each user, RPMs built by each host, tasks submitted by each user, tasks completed by each host, succeeded/failed/canceled builds, number of builds in each target and cluster health.

The information in these reports is useful, but its format is not easy to manipulate. It cannot be used by external monitoring software, like Grafana, and also misses valuable information about the Koji builders that can help to detect problems in advance and/or react promptly and accordingly to the issue.

3 Prometheus - Improving Koji's monitoring

Prometheus is the monitoring tool chosen to improve the quality of the Koji service. It is a highly-reliable open-source tool written in Go. It has a lot of advantages, mainly, it is easy to use, is flexible, has a good performance, uses a pull-based model, and it is quite popular and has good feedback from the community.

In general, Prometheus scrapes metrics data from HTTP endpoints and then pushes that data into a database that uses a multidimensional model. The idea of this project was to expose the Koji information on an HTTP endpoint that can be scrapped by Prometheus servers, as shown in Figure 2.

Figure 2. Implemented Prometheus architecture on Koji system

Prometheus works taking in account these three points:

- **Data collection and retrieval:** It follows a pull-based model, where it periodically scrapes data from Koji web. Koji web script was created to gather relevant information from Koji using Prometheus metrics and exposes it under https://koji.cern.ch/metrics, allowing Prometheus servers to gather that information frequently.
- **Data storage:** The collected metrics are stored in a time-series database, providing a historical record of system performance over time.

- **Service discovery:** Prometheus utilizes discovery mechanisms to ensure that new instances are automatically detected and monitored without manual intervention.

3.1 Prometheus metrics types

Prometheus has a python client library that offers four metric types. For the purpose of this project, only two of them were applied to get the Koji metrics that were needed to ensure the health of the Koji service. The two Prometheus metric types are:

- **Counter:** A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart. This metric is useful to get the number of Koji tasks that were completed, succeeded and failed on each day.
- **Gauge:** A gauge is a metric that represents a single numerical value that can arbitrarily go up and down. The value of the capacity or load of each Koji builder node can be taken using that metric.

3.2 Prometheus metrics exporter

One of the challenges was to understand how the Linux team could incorporate a Prometheus metrics exporter into Koji. As Koji is written in Python, the ideal Prometheus client library to use was the Python library.

The first step was to create a Python script to generate Prometheus Koji metrics, called `kojiexporter.py` that can be accessible in `/usr/share/koji-web/lib/kojiweb/` of a Koji web node, and written under `/www/lib/kojiweb/` Koji project path. The patch of the Koji metrics exporter script can be found in https://gitlab.cern.ch/linuxsupport/rpms/koji/-/blob/master/src/prometheus-metrics.patch.

The script is composed by two Python classes:

- **KojiMetrics:** It creates a Koji session, then it gets the list of hosts, channels, and tasks. Based on that information, a collect metrics function was defined to get the Prometheus metrics using the two metrics types explained in section 3.1.
- **PrometheusExpositor:** Responsible for exposing metrics to Prometheus.

To make the metrics available on Koji web interface under a `/metrics` tab, a new function, called metrics, was added to the `/www/kojiweb/index.py` file. This function will call the Koji exporter script and expose the data. The patch is available in https://gitlab.cern.ch/linuxsupport/rpms/koji/-/blob/master/src/prometheus-metrics-index.patch.

The two patches were added to the Koji spec file, as well as the `python-prometheus_client` python package, as a requirement.

The Prometheus metrics defined by type were the following:

The Gauge Koji metrics:

- `koji_builders_update`: Returns the timestamp of the last update by host, and it also gives the information if the Koji builder is enabled and/or ready.
- `koji_enabled_hosts_capacity_per_channel`: Returns the capacity of hosts by channel.
- `koji_enabled_hosts_count`: Returns the number of hosts by channel.
- `koji_hosts_enabled`: It is a boolean and checks if the hosts are enabled.

- `koji_hosts_ready`: It is a boolean and checks if the hosts are ready.
- `koji_hosts_capacity`: Returns the capacity of each enabled host.
- `koji_enabled_hosts_capacity_per_channel`: Returns the capacity of enabled hosts per channel.
- `koji_task_load`: Returns the task load per host.
- `koji_packages_per_tag`: Returns the number of packages per tag.
- `koji_builds_per_tag`: Returns the number of builds per tag.
- `koji_builds_per_package`: Returns the number of builds per package.

  The Counter Koji metrics:
  Returns the number of Koji taks per channel, method, and tag.
- `koji_waiting_tasks`: Returns the tasks waiting/unscheduled.
- `koji_in_progress_tasks`: Returns the tasks in-progress.
- `koji_task_completions_total`: Returns the tasks completed.
- `koji_task_succeed_total`: Returns the tasks succeeded.
- `koji_task_errors_total`: Returns the tasks failed.
- `koji_tasks_total`: Returns all the Koji tasks.

3.3 Upstream Challenge

Before starting this project and creating the Prometheus metrics for Koji, a discussion was opened in https://pagure.io/koji/issue/3812 to get feedback from the Koji maintainers.

The idea was also to contribute to upstream Koji, and make the metrics web method a native part of Koji. That way, Koji users could benefit from it as well. The Koji upstream feedback was that this kind of monitoring tends to be highly specific to a deployment. They decided that it doesn't belong to upstream Koji. It is understandable that the Prometheus metrics built, can be very specific for the CERN use case.

Then the second idea was to build a web plugin that could be installed to make Prometheus metrics available, then it would make it optional. Unfortunately, there is no concept of web plugin in Koji.

The final option was to create our changes as a local patch we would maintain on top of upstream Koji. The patch link with the code implementation was sent to the upstream discussion above, because we believe that it can help other Koji users that have the same need.

4 Grafana - Koji's metrics visualizations and alerting

Grafana is an open source analytics and interactive visualization web application. At CERN, this monitoring system is used to produce dashboards and alerts. One of the supported data sources is Prometheus, that can be easily defined and then based on each metric, dashboards were created.

In Figure 3, there are four visualizations: the number of Koji builders enabled; the information about each Koji builder: the hostname, if they are enabled and ready, and the last update of each host; The pie chart represents the number of total tasks, and how many succeeded and failed for the current day; The last visualization is the number of waiting and in progress tasks in the last five minutes.

The number of hosts and the host's capacity by channel can be visualized in Figure 4, as well as the current load and maximum capacity by each enabled Koji builder.

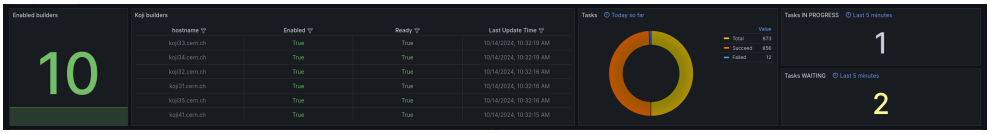

Figure 3. Koji's dashboards

Figure 4. Koji's dashboards

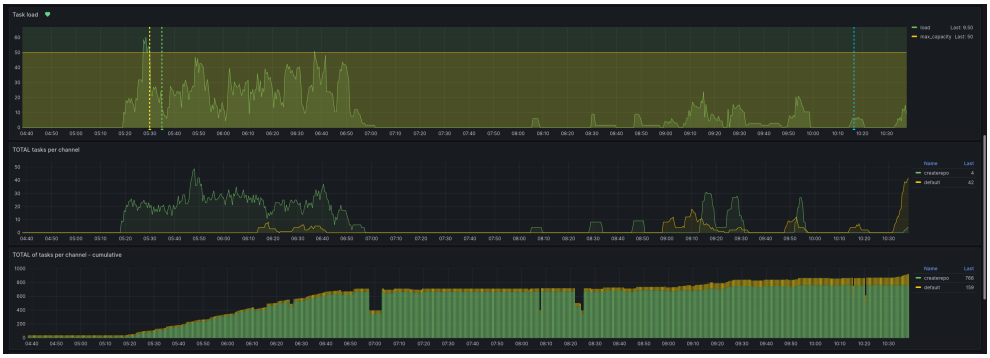

Figure 5. Koji's dashboards

In Figure 5, three time series were defined: the sum of task load of the hosts in each moment, which helps us to visualize Koji activity over time. The other two dashboards represent the number of tasks and the cumulative number of tasks per channel.

These visualizations help to monitor the state of the Koji system, and the big advantage is having this information consolidated in a single place.

4.1 Alerts

The dashboards are useful, but we are not looking at them all the time. Fortunately, in Grafana, there is also the possibility to create alerts and integrate them with Mattermost. The alert rules created were:

- **Task load**: It is triggered when the Koji builders are reaching the max capacity available. This is important to understand if the hosts are overloaded, and if that is the case, one solution can be to add more Koji builders.
- **Koji builder giving updates but disabled**: If there is a host that is not enabled but is still alive, meaning that the last update happened in the last minute, the alert will be triggered.

For us, it is important to understand what is happening with the service when we disable a node, but it still reports activity.

- **Koji builder enabled and ready but not giving updates in the last 60s**: Conversely, if a node is ready and enabled, it is important that it gives updates frequently. Otherwise, it means that the service on that node is down and not reporting correctly.
- **Koji builder enabled but not ready**: As mentioned before, if a Koji builder node is enabled but 'kojid' service is not running, then the node is not ready to receive tasks. This is a very sensitive case, and it is important to detect it if it happens. One of the implications is the load growth on the other nodes. After building this alert, some discoveries were made. We learned that if a node is overloaded, it changes its state to not ready, in order to signal to the hub that it can't handle more new tasks. Recently, one of the challenges was to adapt this alert to not be triggered when the node is overloaded.

5 Conclusion

Now we extract Koji metrics using Prometheus through a Koji web method and expose that information under https://koji.cern.ch/metrics. It was practical to plug in that information into the Grafana monitoring system.

Nowadays, it is easy to keep track of the Koji monitor system via the visualizations and alerts created. In one place, it is possible to check: How many Koji builders are enabled and ready? How many tasks are being done? How many successes and failures? All these questions are now simple to answer.

The capacity of the Koji builders is reasonable and doesn't reach the max load in moments of peak activity.

Since the metrics generated can be specific for the CERN use case, this work is not a native part of Koji. The implementation work can help other people with the same goal and benefit from it. We feel that Prometheus Koji's metrics make Koji easier to monitor and may benefit others.

References

[1] Koji's documentation https://docs.pagure.org/koji/, [Online; accessed 27-August-2024]

[2] Prometheus Documentation https://prometheus.io/docs/introduction/overview/, [Online; accessed 23-August-2024]

[3] Grafana Labs Grafana Documentation https://grafana.com/docs/grafana/latest/, [Online; accessed 23-August-2024]