A reverse proxy for the Docker API

Anthony Baire
Université de Rennes 1 / UMR IRISA
Campus universitaire de Beaulieu
263 Avenue du Général Leclerc - CS 74205
35042 Rennes Cedex

Summary

As part of the Inria Continuous Integration service, we deploy Docker executors to manage a large number of small projects. This service also uses Jenkins and GitLab-CI.

The Docker solution was chosen because it is easy for our users to adopt and because it allows the service to be resized on demand. In addition, the stable API gives us a point of control to optimize and reconfigure the platform without having to involve users.

We have developed a reverse proxy for the Docker Engine API. Beyond the basic functionalities (authentication and access control), we have experimented with more advanced uses:

- *application of policies (injection of parameters into the container configuration, prohibition of the use of special options, etc.);*
- *isolation of users by using namespaces;*
- *transparent cache of HTTP and HTTPS connections, in order to reduce traffic with the archives of popular packages (Debian, Fedora, Maven, etc.). The HTTPS cache is made possible by installing a temporary certification authority inside the containers when they are created.*

The tool is developed in asynchronous, expandable python3 (new authentication policies and methods can be added as plugins) and integrated into the boot2docker image (used by docker-machine) to facilitate its deployment on a cluster of machines for GitLab-CI.

This tool has been tested on our qualification platform since October 2018. We will be using it very soon on our production platform. We will quickly present the various functionalities of the tool as well as feedback on its use.

Keywords

Continuous Integration, Docker, GitLab-CI, Proxy, Caching

1 Introduction

In the context of the Continuous Integration service at Inria we are setting up Docker executors that are expected to handle efficiently a large number of small projects, without breaking users habits.

We developed a reverse proxy for the Docker engine API. Along with basic features (authentication & access control), we implemented namespaces to isolate user's resources from each other and we are experimenting advanced usages:

- transparent caching of HTTP/HTTPS traffic;
- restricted access to the Docker API from within a container.

2 Context

2.1 Continuous Integration at Inria

These developments are made for Inria's Continuous Integration service[1]. This is a common infrastructure usable by all research teams at Inria (and its partners) to improve the overall quality of software development. It has been running since 2012 and as of March 2019 we are hosting 390 Jenkins instances and 700 virtual machines in a private cloud.

As we are in a computer-science research context, some distinctive facts have to be taken into considerations:

- most of projects are small (few developers, low or irregular activity);
- developments are mostly research-oriented, many developers do not have a strong experience in software development (eg: PhD students, junior engineers, ...)

2.2 Issues

In the current set-up of this service, users are expected to maintain their own executors (hosted in virtual machines) for running build jobs with Jenkins or GitLab-CI. This has several implications:

- *suboptimal resource allocation*: virtual machines are created on a per user basis (or per project), many are idle most of the time and they cannot scale up when the project has a spike in activity (eg. during a coding sprint);
- *maintainance burden*: users have to administrate their VMs (and some may not dot it very diligently);
- *reproducibility*: users may install additional software on the VM or update its configuration ; if they do not carefully automate these steps then the knowledge gets lost

1. hosted at https://ci.inria.fr

Finally the Inria CI service is well suited for build-intensive projects, where users have incentives to tune well their configurations and make an efficient use of the resources; but hosting small/low-activity projects is suboptimal.

2.3 Objectives

Our main objective is to provide shared CI runners usable with GitLab-CI and Jenkins and suitable for small projects (and bigger projects to some extent). Especially we have five requirements in mind:

- *resources sharing:* computing resources should be allocated only for the duration of the job and released immediately after;
- *caching*: as build jobs imply repeatedly downloading lots of dependencies, the solution should allow efficient caching;
- *bootstraping*: build environment should be easily scriptable, to remove the burden of administrating permanent virtual machines and improve reproducibility;
- *ease of adoption*: our user base mainly consist of researchers, who may not be familiar with state-of-the-art software development tools and practices. The solution should not deviate much from their development environments;
- *stability and maintainability:* we wish to maintain the service in the long run. The solution should rely on stable and proven tools and APIs.

2.4 The Docker solution

Docker quickly appeared to be the most suitable solution for our needs. It is supported by GitLab-CI and Jenkins, also most of our requirements are fulfilled out-of-the-box:

- containers can be created on-demand for the duration of a job, thus achieving good resources sharing;
- Docker images are easy to build thanks to the integrated builder and its *Dockerfile* syntax, the development environment is easy to bootstrap;
- Docker is nowadays widely adopted in the research community, it is abundantly documented and easy to install on the developer's workstation, which makes it easy to adopt by our users;
- the Docker engine is distributed under a free software license, it has a stable API, and has a large user community, which gives us good confidence in its long-term stability and maintainability.

However switching to Docker comes with its own issues:

- *security*: granting access to the Docker socket is equivalent to granting root access on the machine hosting the Docker engine;
- *caching*: running CI jobs inside a temporary container undermines the caching strategies of package managers, dependencies are downloaded again for every new job unless users make configuration efforts to store the cache in an external volume.

2.4.1 Access to the Docker socket

Although giving access to the Docker socket is not strictly needed for running continuous integration jobs, we think it is an essential feature in our context because we want to support building images and because we prefer to stick as close as possible to our developer's environment.

- Building Docker images and pushing them to a registry requires access to a Docker engine. Hosted projects should be able to set up a full deployment chain that includes building a specialised image for their jobs and building the final images to be shipped to the end users.
- Test setups may require additional services (e.g. database servers, message brokers, ...) to be deployed, which is commonly achieved by running additional containers. Without access to the Docker socket, users have no choice but to use platform-specific configurations[2], especially they cannot just reuse the *docker-compose* configuration from their development environment. While this point may look like a very minor issue, it is actually very relevant in an academic context, where developers do not have strong incentives to set up proper continuous integration (in comparison with the software industry). Moreover they tend to work on many small projects in parallel and may no be willing to spend time on the configuration for each individual project. Staying as close as possible to the developer's environment reduces setup costs and increases adoption.

We identified two approaches to provide access to the Docker socket:

- isolate each user within a virtual machine;
- implement some filtering in the Docker API, to block privileged actions and prevent users from accessing resources of other users.

2.4.2 Caching

Efficient caching is itself a challenge because of the copy-on-write nature of Docker containers.

A Docker image is technically an immutable snapshot of a container filesystem. When a new container is created from this image, its filesystem is layered over the image using a copy-on-write technology[3]. Any write in the container's filesystem remains local to this container and the original image always remains unchanged. When a container is destroyed, its local changes are irreversibly dropped.

This copy-on-write strategy is good for reproducibility as it guarantees that jobs are always started in the same pristine environment. However this also wipes out the package manager caches unless the user takes specific actions.

This issue can be worked around by storing the caches in external volumes, but not without inconveniences:

2. For example in GitLab-CI, these additional containers can be declared in the services configuration key. https://docs.gitlab.com/runner/executors/docker.html#the-services-keyword.

3. By default dockerd uses the OverlayFS union filesystem from the Linux kernel.

– the configuration would depend on the package manager used in the job, as each package manager uses a different cache directory (e.g. apt uses */var/cache/apt*, maven uses *~/.m2/repository*, …);

– as we intend to deploy multiple docker hosts, subsequent builds may not be scheduled on the same host. To be useful the external volume would have to be mounted from a remote location, which implies a more complex configuration;

– external volumes cannot be mounted during image builds[4] (from a Dockerfile);

– for obvious security reasons external volumes hosting caches cannot be shared between users, which decreases their overall efficiency.

Requesting our users to adjust theirs configurations in a complex manner is definitely not an option. This would increase the occurrence of configuration errors and support requests and more generally increase user frustration. Also most users would likely just ignore these recommendations. This is typically a kind of *tragedy of the commons*. The shared resource is efficiently used only if users cooperate actively, but individual users will not get immediate benefit from their efforts.

Note that regarding image builds, Docker mitigates the problem by caching the intermediate images committed after each step in the Dockerfile. It enables quick iteration during development, but it is less adequate in continuous integration because the cache manager ignores the side-effects of 'RUN' commands (which may cause false positives and false negatives).

From these observations, we acknowledged that implementing a useful cache requires tuning the configuration of every container in the platform. This is achievable if we have a way to inject configuration changes transparently (i.e. without user intervention).

3 The Tool

3.1 Focus on the Docker Engine API

The Docker Engine API[5] is the interface between the *docker* client command and the *dockerd* daemon that actually runs the containers. It is a REST API defined over HTTP, except for a limited number endpoints which require protocol upgrades[6]. By default the Docker engine is configured to provide this API over a unix socket bound to `/var/run/socket.sock`, but it is possible to provide it over TCP either. IANA assigned TCP port numbers 2375 and 2376 for that purpose[7].

While API is designed to control every aspect of the Docker engine, early in the project it was clear to us that the configuration details of our infrastructure should be abstracted. The users should use the API the same way they would do on their workstation, and the

4. This might no longer be true in the future because the new buildkit backend (which is being integrated in docker) provides an experimental syntax for that purpose: https://github.com/moby/buildkit/blob/b939973/frontend/dockerfile/docs/experimental.md.

5. https://docs.docker.com/engine/api/latest/

6. These are the endpoints for attaching to the container console (`docker attach`) or to an additional terminal (`docker exec`). Theses features require bidirectional streaming which is not available in regular HTTP. The API implements them either by upgrading the connection to a Websocket or by hijacking the underlying TCP stream.

7. Port 2375 for plain text sockets and port 2376 for TLS sockets.

configuration details for tuning their container to our infrastructure should be added transparently by us.

We thus decided to develop a HTTP reverse proxy. This tool would stand between the Docker engine on one side and the client on the other side (a GitLab runner, a Jenkins instance or any command run by a user job). This tool could address multiple purposes, for example:

- implement authentication and access control (eg: restrict privileged commands, isolate user in namespaces);
- inject additional configuration options in container creation requests;
- balance the requests over a cluster of docker nodes.

3.2 Introducing CTproxy

We developed a tool named CTproxy[8], which stands for "Container Tuning Proxy".

It is a reverse proxy that understands the Docker Engine API. It is written in Python3 and its design is fully asynchronous (based on the aiohttp framework).

As the project was exploratory in its nature, we implemented various features to try different possible configurations:

- **Multiple listeners**

  It can be configured to listen on multiple sockets, each listener may have a different configuration.

- **TLS**

  Listeners can be secured with TLS, with optional validation of the client certificate.

- **Authentication**

  Client authentication may rely either on the HTTP Authorization, TLS client certificate or remote user id (for unix sockets).

  Authentication modules are implemented as plugins. Thus any authentication source can be supported by implementing the appropriate plugin.

- **Policy enforcement**

  Policies are implemented as plugins, and a listener may be configured with 0 or more policies.

  Policy plugins can be used for two purposes:

  - implementing access control;
  - injecting additional parameters in Docker commands.

8. https://gitlab.inria.fr/inria-ci/ctproxy

The tool comes with a built-in policy named 'conservative' which blocks all Docker commands or container options that elevate privileges (eg: --privileged, --cap-add, ...).

– **Transparent caching**

  The transparent cache feature, when enabled, alters the configuration of containers created through the Docker API. A selection of domains to be cached are redirected towards a local HTTP caching proxy (using docker's --add-host option) and a custom certificate authority is installed inside the container to allow caching the HTTPS traffic as well. The custom CA is generated with a short lifetime and rotated automatically.

  This feature is intended to be used with bandwidth hungry sites. In our continuous integration context, these will be the most popular package repositories (eg: Debian, Fedora, Maven, ...).

– **Namespaces**

  When namespaces are enabled, all resources names (containers, images, volumes, ...) are mangled to provide isolation between users.

  On the implementation side most resource names (containers, volumes, ...) are just prefixed with the namespace, whereas the mangled images use a more complex format to allow push/pull operations (the registry API requests go through another reverse proxy which demangles the name and forward them to the original registry).

– **Temporary listener**

  The purpose of temporary listeners is to provide a limited access to the Docker API from inside a container running a GitLab-CI job. This allows users to build and push Docker images without having to provide their own dedicated runner.

  When a container is created, a new unix socket is created and mounted at /var/run/docker.sock inside the container and this socket is associated with a temporary namespace. When the container terminates, all resources created through this listener are destroyed.

4 Deployment and future works

At the time of writing, most of the developments are completed. Experimentations have been carried with GitLab-CI and Jenkins.

4.1 GitLab-CI

We have been running three experimental GitLab-CI runners since 2018. They are made available on a voluntary basis to users with an Inria account.

The runners are using the docker+machine executor in an autoscale[9] configuration: virtual machines are created on-demand for hosting the Docker daemons that run the jobs.

The runners are configured to run each job in a dedicated VM and to make the Docker socket accessible from inside the container running the job (mounted as an external volume). As the jobs can use the Docker socket, we must assume that their owners have root access to the VM. Thus we rely on the hypervisor to isolate jobs from each other.

We use CTproxy to implement transparent caching. We integrated it within the boot2docker image to make it deployable by the docker+machine executor.

We deployed a nginx server to act as the caching proxy, in addition to the traffic redirected by CTproxy, it also caches all accesses to the official Docker registry (registry-1.docker.io).

So far around 8000 jobs from 115 different projects were processed and the cache served 225000 requests with 85% cache hits and including 30% of requests over TLS.

Future works will focus on scaling (switching to Cloudstack) and better metrology.

4.2 Jenkins

Experimentations with Jenkins have been carried out as well, but they are currently on hold because our efforts are focussed on GitLab and because we have not yet settled on a scaling strategy.

5 Other contributions

During the development of our reverse proxy, we made additional contributions to the community:

- 1 bugfix (#693) in the Docker plugin of Jenkins;
- 1 bugfix (#988) in the docker-java api package;
- aio_gnutls_transport: a GnuTLS-based transport for the python3 asyncio framework. Unlike python's native TLS transport it supports half-closing TLS connections.

6 Acknowledgements

We would like to thank Christophe Demarey, Florent Paillot and Patricia Bournai for their support and suggestions throughout this project. More generally we would like to thank the support team of the continuous integration service and the IT department at Inria.

9. https://docs.gitlab.com/runner/configuration/autoscale.html