Continuous Integration of Robotic Software

Lukas Graber

August 10, 2018

Contents

Introduction

The objective of the project was to build a system for Continuous Integration of Robotic Software across a cluster of robots. In addition, information of the robots in the cluster should be collected and their status be monitored. For the last part we reused an already existing monitoring solution.

Prerequisites

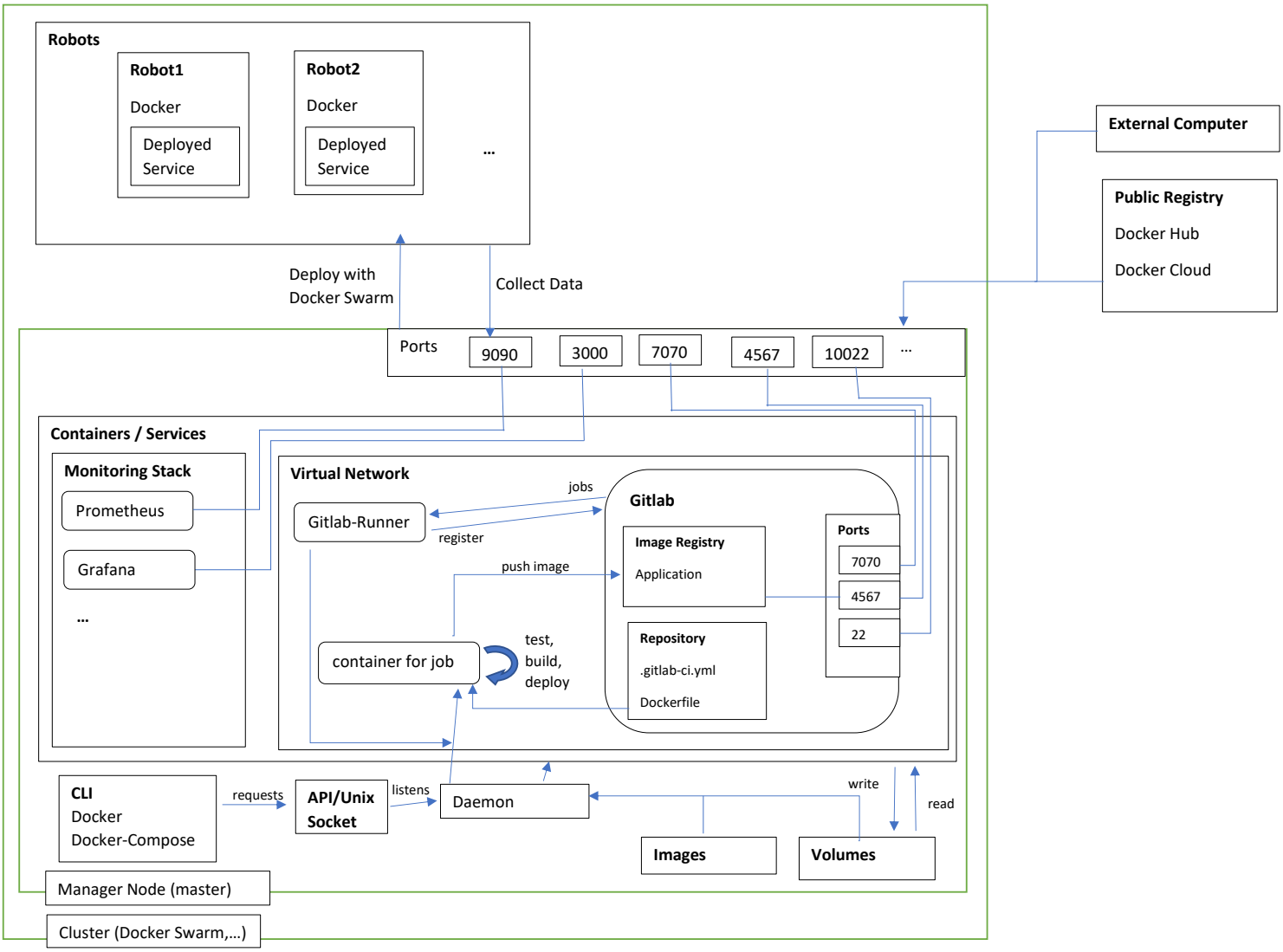
For the final system to work properly, each computer that will be part of the cluster must have **Ubuntu** installed (or at least simulated on a virtual machine). Furthermore all robots/computers must have the **Docker Engine** installed on them. At least one machine should also have **docker-compose** for the **Gitlab** and **Gitlab-runner** instance. To access the Gitlab instance later, all machines in the cluster should also be connected to the same router.

Docker

Our whole system relies heavily on the Docker Engine. Docker makes it possible to define an application in one specific way and it is guaranteed to behave the same, independent of the underlying OS. Because there are multiple components involved in our system, each of them lives in a different docker container. Furthermore, these containers should be able to communicate with each other. That is the reason why they need to be on the same network. Instead of initialising every container by hand and putting them on the same network manually, we have used docker-compose, a container orchestration tool.

System Overview & Components

The following image gives a first overview of our final system setup.



Our final system consists of five major parts:
1.) Gitlab
2.) Gitlab-runner
3.) Gitlab Image Registry
4.) Cluster orchestration
5.) Continuous Integration (CI) pipeline
In the following paragraphs, it will be explained how the different components of our system were configured and how they work together.

Gitlab and Gitlab runner

In the docker-compose file we defined the Gitlab service and Gitlab-runner service and their configuration. Docker-compose will create two containers and put them on the same network. The following lines show how we configured these two services. One thing to mention is that each service can only run one image.

```yaml
version: "3"

services:
  gitlab:
    image: 'gitlab/gitlab-ce:latest'
    restart: always
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        external_url 'http://master:7070'
        gitlab_rails['gitlab_shell_ssh_port'] = 10022
        registry_external_url 'http://master:4567'
    ports:
      - '7070:7070'
      - '10022:22'
      - '4567:4567'
    volumes:
      - '/srv/gitlab/config:/etc/gitlab'
      - '/srv/gitlab/logs:/var/log/gitlab'
      - '/srv/gitlab/data:/var/opt/gitlab'

  gitlab-runner:
    image: 'gitlab/gitlab-runner:latest'
    restart: always
    volumes:
      - '/srv/gitlab-runner/config:/etc/gitlab-runner'
      - '/var/run/docker.sock:/var/run/docker.sock'
```

Gitlab service:
The Gitlab service is the center of gravity of our system. This service hosts the project repository, it defines the CI pipeline, it stores images in its own registry and it is even possible to integrate a **Kubernetes** cluster. We have used a special Gitlab image, which is a prebuild image from **DockerHub**. If this image would not be available in the local registry, it is pulled from DockerHub. Furthermore, we wanted to make sure that the container will restart always if it has been down. By setting the Omnibus environment, we define specific configuration options for our Gitlab instance. These changes will be written to the gitlab.rb file.
Because we want to access the Gitlab instance from within a browser, we need to set it to an external url. The Gitlab instance should also be accessible from other computers. That is why we need to use the ip that the router assigns to the machine hosting the Gitlab instance instead of using localhost. This makes it possible for all machines connected to the same router to access Gitlab over the external url which was set in the docker-compose file. Instead of using the real ip to access Gitlab, we mapped the ip to some artificial name in /etc/hosts (e.g. master). In addition to that we set the port number to 7070 (Prometheus is later running on port 9090). Thus, the external url contains the name resolution of the physical ip that the router assigns to the machine hosting the Gitlab instance and the port on which this container is sitting. This setup makes it possible for other machines on the same subnet to access Gitlab by entering the external url of the docker-compose file in their browser. Moreover, we set the ssh port to 10022 to enable ssh authentication. Furthermore, we enable the image registry by setting another external url with a different port.
Afterwards, the real physical ports of the host machine are mapped to virtual ports of the container. Because we do not want to block the physical port 22, we have used port 10022. Over the lifetime of a container, the container produces data. To make this data persistent, we had to define volumes. Volumes are stored in a part of the host filesystem which is managed by Docker and can be mounted into a container.
If the Gitlab container is running in a healthy state, other computers can sign in to this Gitlab instance and be assigned to a project by the administrator. Afterwards, the registered machines share the same repository and therefore the same code base.

Gitlab-runner:
The second service is the Gitlab-runner service. This service is configured with a Gitlab-runner image and is mounted to the **Unix Socket**. This is the Socket the **Docker Daemon** is listening to and therefore it will be possible to send requests to this API and create containers from within a container. The other volume will track the config.toml file of the gitlab-runner container, which stores the registered runners and their configuration data in the host machine.
If both services are in a healthy state, it is time to register the Gitlab-runner to the Gitlab instance. The registration information will be written into the config.toml file as follows:

```
[[runners]]
  name = "another-runner"
  url = "http://gitlab:7070"
  token = "62551d3e7fc1fc62f303ac7f45dca0"
  clone_url = "http://gitlab:7070"
  executor = "docker"
  [runners.docker]
    tls_verify = false
    image = "gitlab/dind"
    privileged = false
    disable_cache = false
    network_mode = "teamproject_default"
    volumes = ["/cache","/var/run/docker.sock:/var/run/docker.sock"]
    shm_size = 0
  [runners.cache]
```

The above graphic is the content of the config.toml file after registering the runner. It contains all the information which was entered during registration. The file stores the given name for the runner (in our case another-runner), the token to authenticate to Gitlab and the environment is build with docker as executor. Furthermore the runner uses a service called dind, which stands for docker in docker (image: gitlab/dind). This makes it possible to create docker containers inside a docker container. To use the Docker Daemon of the host machine, we map the Unix Socket to the Socket inside the container. Therefore the containers that the runner will spin up will not be inside another container but like the container itself on the host machine.
After registering the runner, we need to add two additional lines into the config.toml file. The first one is the clone url, which overwrites the url for the Gitlab instance. This is used if the runner is unable to connect to Gitlab on the url Gitlab exposes itself. The second line to be added, is the network mode, which needs to be the same network on which the Gitlab instance is sitting. Otherwise the runner instance will be put on the default docker network.
The runners will later play a key role in the CI pipeline.

Gitlab Image Registry

The application we want to deploy onto the cluster will be built as a docker image. To define an image we use a Dockerfile, in which we specify layer by layer what should be included in our application. Because we will deploy our application as docker images, these images need to be stored at a place that is easily accessible during the execution of the pipeline. That is why we were using the Gitlab image registry that comes integrated with Gitlab. To activate the registry, we had to change some configuration in the docker-compose and daemon.json file.
Later, we will build the image based on a Dockerfile, push it to the registry and deploy it on the machines in the cluster, all from within the pipeline itself.

Cluster Orchestration (Docker Swarm)

In our final system setup, we use a cluster orchestration tool to deploy an application onto a cluster of robots/machines. We had the choice between Kubernetes and Docker Swarm. In the end, we decided to use Docker Swarm to manage our cluster of machines. The reasons for this decision are explained later on.

CI Pipeline

The pipeline is the place, where everything comes together and Continuous Integration is achieved. It is controlled by the .gitlab-ci.yml file. This file defines the different jobs on different stages that the runner should work on. When pushing the file to Gitlab a pipeline gets triggered and the pipeline is set to a pending state. Then, the runner will pick up job by job and starts working on the specified tasks. If an error occurs, then the pipeline will stop right away. With specific tags it is even possible to control specific runners.
This is the .gitlab-ci.yml file for our Continuous Integration system:

```
stages:
  - test
  - build
  - deploy
variables:
    IMAGE_TAG: "master:4567/root/firstproject"
before_script:
    - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN http://master:4567

build_image:
  tags:
    - another-runner
  stage: build
  script:
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG

deploy_image:
  tags:
    - another-runner
  stage: deploy
  script:
    - docker service update --image $(docker inspect --type image --format '{{index .RepoDigests 0}}' $IMAGE_TAG:latest) ello

  environment:
    name: $CI_ENVIRONMENT_NAME
```

In the file above, we define three stages and set a variable which contains the name of the image in the registry we want to deploy. Before any script will be executed we log into the registry with a predefined Gitlab token. This makes it possible to pull and push to the registry from within the pipeline itself. The first job will build a newer image based on a Dockerfile located in the repository in Gitlab and push it to the registry. The second job will deploy this image by updating an existing service (e.g. ello). This means that the service will not be destroyed but only the container it is host of. At the same time a container with a newer image will be started on each machine in the cluster.

Final System Setup & Workflow

In our final system setup, we had one laptop running the Gitlab and Gitlab-runner instance. Another machine created the cluster and promoted the first machine to manager status. We also had some more workers in the cluster to deploy to.
We pushed an image to the Gitlab registry and deployed a service with this image over the nodes in the cluster. This service can be updated from within the pipeline itself. Furthermore, we had a Dockerfile, additional files defining our application and a .gitlab-ci.yml file uploaded to Gitlab.
Each time a computer of the cluster updates the repository on Gitlab, a new pipeline gets triggered. A runner will pick up and work on the different jobs specified in the .gitlab-ci.yml file. First, it will log in to the image registry, build a new image based on the Dockerfile and push it to the registry. In another stage, it will update an already existing service and therefore deploy the latest image on each available machine in the cluster. Afterwards a newly build application is deployed on each machine in the cluster. Therefore, Continuous Integration is achieved.

Monitoring

Our monitoring system makes use of Prometheus, a systems monitoring and alerting toolkit. Instead of building our own monitoring system, we have used a very sophisticated solution called Swarmprom. Swarmprom had the whole monitoring system already build. This meant that we just had to clone the repository and afterwards deploy the stack over our cluster. The only thing we had to change was the port on which the Gitlab instance was running, because it is now also used by Prometheus.
After some time, all the needed containers were started on the worker nodes and reported to the Prometheus instance, which was only running on the manager machine. In Prometheus, all the data is stored in a database and it is possible to access this data with the query language PromQL.
After deployment the different services of the stack will then expose the collected data to Prometheus. This Prometheus instance can be accessed over the browser from all machines on the same subnet. With Grafana, it was possible to represent the Prometheus data graphically by inserting a PromQL query into Grafana.

Possible Areas of Application

- The final system setup describes a workflow on how to deploy any application onto a cluster of remote computers. By defining a robotic application as an image, a deployment system especially for robotic software comes into being.
- With our system setup, we achieve Continuous Integration. As soon as one file changes in the central repository in Gitlab, a pipeline gets triggered and a new image gets deployed on all robots in the cluster. If the pipeline passes, it is guaranteed to work on the different robots properly, especially because the application is run in containers and therefore behaves the same no matter what the underlying OS looks like.
- The Gitlab instance brings the development, the building, the testing and the deployment of an application together in one place. The pipeline gives immediate feedback to the developer and the results can be seen after the deployment inside the different containers.
- The next step would be the deployment of a working turtlesim application. By changing the image, we could change the movement of the turtle for example. After deployment, this change would be visible inside the container on each machine of the cluster.
- The monitoring system enables to get information about the health of the robots in real time. This makes it much easier to diagnose the status of the robots and the cluster.
- By configuring the setup of the containers, it will be possible to control real hardware on the host machine. This makes it possible to connect to the external display from inside a container for example.
- Docker Swarm makes sure that an application is running on each machine in the cluster. If a robot was temporarily down or a container was shut down, the service will be automatically redeployed onto the machine by Docker Swarm.
- If the Gitlab instance would not be bound to the router, it would be possible to access it from many different places. A widely distributed team could therefore work on a robotic application and immediately build, test and deploy it. With the monitoring system, people could also access the Prometheus or Grafana instance and therefore monitor the consequences of the deployment.
- By extending and polishing some features, the system would become much more robust. In the end, this system might be a blueprint for designing a CI system for robotics with additional monitoring for professional use cases.

Difficulties & Solutions

During the project, we encountered the following difficulties along the way:

- At first, we went with Kubernetes as the cluster orchestration tool. Unfortunately, the registration of kubectl in the pipeline itself was not working properly. Therefore, we used Docker Swarm for our final system.
- The pipeline is controlled by the .gitlab-ci.yml file. For the deployment to work, we needed to update our service. We had to use a special line in the .gitlab-ci.yml file to make the latest version of our image deploy to the cluster. Otherwise, the latest image will not be deployed.
- One of the machines in the cluster, which was also hosting the Gitlab and Gitlab-runner instance, had the development version of docker installed. Every time we tried to deploy over this machine as swarm leader, the deployment was breaking all the Docker Daemons of the other machines in the cluster. The solution was to create the swarm on another machine and promote the machine with Gitlab running on it, to manager.
- For the deployment to work properly, every machine needs to have the ip of the machine hosting the image registry mapped to an artificial name in /etc/hosts. Otherwise, the actual deployment over the script would not work.
- For our demonstration of the system, we wanted to create a turtlesim application. Unfortunately, the container was not able to connect to the external display of the host machine. A solution would be to start the container with a special configuration option that allows to connect to external display.

Room for Improvement & Missed Opportunities

Router Scope

At the moment, we can only operate if we are on the same subnet because the system is dependent on the router. Furthermore, our mapping in /etc/hosts is bound to the ip which was assigned by the router. Therefore, if the lease is renewed and the router assigns a different ip, we need to remap ip and name in /etc/hosts again.

Real Robotic Software

Up till now, we have only changed text files and deployed the resulting images across our cluster. The next step would be to use robotic software and test applications on actual robotic hardware. What we tried so far was to deploy a turtlesim application onto our cluster. There was still an issue with connecting to the display, but that could be solved over some configuration option while setting up the container.

Swarmprom vs Own Monitoring Solution

In our final system setup, we have been using an already existing monitoring solution called swarmprom. Instead of using swarmprom, we could have created our own monitoring system from scratch. Because of the approaching deadline, this was not possible. One problem with swarmprom though was that it was recording all but one of the computers of the cluster.

Kubernetes vs Docker Swarm

In our current version of our system we are using Docker Swarm instead of Kubernetes. Although Kubernetes might be the more sophisticated tool for professional use, it produced just too many inconveniences. First of all, Kubernetes can only operate when turning swap memory off. But this means that after some time the machine will get very slow and inefficient. When considering the fact that our system should create a more efficient workflow for the deployment of robotic software, this can not be overlooked. The second issue was the fact that the registration of kubectl in the pipeline was not working properly because kubectl was not able to find the DNS server. Docker Swarm on the other hand comes integrated with the Docker engine and can therefore be controlled by runners that use the service dind and are mounted to the Unix Socket of the host machine.
But Kubernetes might still be a more sophisticated cluster orchestration tool and might be working for other people. One of the advantages that Kubernetes has to offer is the fact that it is integrated with the omnibus environment of Gitlab. Furthermore, Prometheus could just be switched on over the Gitlab interface.

Docker Version Conflict

One factor that made our system much more unstable was the fact that the computer on which the Gitlab instance was running used a development version of the docker engine. This computer was meant to be the swarm leader. But whenever the computer tried to create a service and deploy it on the other machines, it was breaking the Daemons of the other machines. Our solution was to create the swarm on another machine and promote the machine with the Gitlab instance as a swarm manager. The other machine created the service and the promoted manager machine was able to update a service with a runner instance. But when doing the monitoring part there have also been some problems when the promoted manager was the leader of the swarm. The monitoring was only working properly when other machines were the leaders of the cluster.
The solution for this would be to install a stable version of Docker Engine on each machine that is supposed to be in the cluster. This would result in one machine hosting the Gitlab and Gitlab-runner instance as well as being the leader of the docker swarm. Consequently, our whole system becomes more cohesive and focused in one place, instead of spreading the whole functionality across the machines in the cluster.

Summary

In the end, we have built a system for Continuous Integration / Continuous Deployment of an application over a cluster of machines. By using real robotic software (and hardware), the system would become a solution for Continuous Integration of robotic software. Furthermore, we have used an already existing monitoring solution to record the status of the machines in the cluster. At the same time, our system might be unstable because of network issues. This work can be a first step on designing a Continuous Integration system of robotic software for professional use cases.