Continuous Integration / Continuous Deployment of Robotic Software

Lukas Graber

7. August 2018

Inhaltsverzeichnis

Introduction

The objective of the project was to build a system for Continuous Deployment of Robotic Software onto 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 solution.

Prerequisites

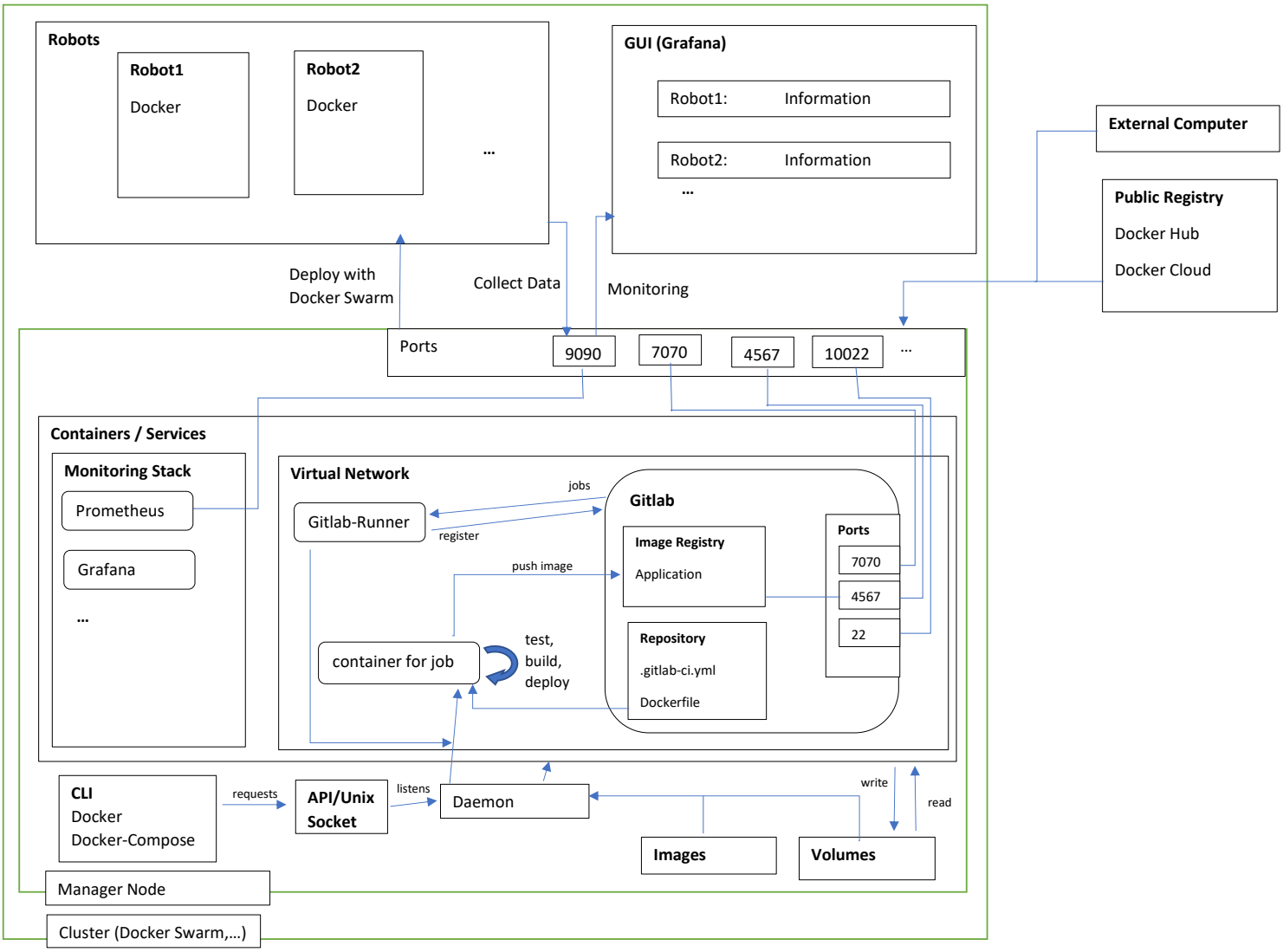
For the final system to work, each computer that will be part of the cluster must have **Ubuntu** installed (or at least simulated on a virtual machine). Furthermore all robots/computers must have the **Docker** engine installed on them for the deployment to work. At least one machine should also have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

Why Docker?

Our whole system relies heavily on the Docker engine. This makes it possible to define a system in one specific way and it is guaranteed to behave the same way independent of the underlying OS. Because there are multiple components involved in our system, each of them is run in a different docker container. Furthermore, these containers should be able to communicate with each other. That is 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 used docker-compose, a container orchestration tool.

System overview

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



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

Gitlab and Gitlab runner

In the docker-compose file we defined the Gitlab service and Gitlab-runner service and its configuration. Docker-compose will create two containers and put them on the same network. The following lines show how we configured the Gitlab and the Gitlab-runner service.

```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'
```

We defined two services, a Gitlab and a Gitlab-runner service. Each service can only run one image.

Gitlab service:

The Gitlab service will be our center of gravity for our system. This service is our central code base by hosting the repository, it defines the CI/CD pipeline, stores images in its own registry and it is even possible to integrate the cluster when using **Kubernetes**. We use the image gitlab/gitlab-ce:latest, which is a prebuild image from **DockerHub**. If this image is not available in the local registry, it will be pulled from DockerHub. Furthermore, we want to make sure that the container will restart always if it is down.

By setting the GITLAB_OMNIBUS_CONFIG variable, 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 manager machine. This makes it possible for all machines connected to the router, which means that they are on the same subnet, to access Gitlab over the external url 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). This setup makes it possible for other machines on the same subnet to access Gitlab, by eiter entering ip:port or master:port (when remapped at other machines) into the browser. Moreover, we set the ssh port to 10022 to enable ssh authentication, which could be useful for making the workflow faster. Furthermore, we enable the image registry by setting another external url with a different port.

Afterwards the real physical ports of the machine are mapped to virtual ports of containers/services. Because we do not want to block port 22 for real, we mapped it to port 10022. At last, we need to define volumes to make any data persistent. The volumes map a host folder to a mount point in the container.

If the Gitlab instance is set up, then other computers can sign in to this instance and wait to be assigned to a project and a rank by the administrator. Afterwards the computers 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, which stores the registered runners and their configuration data in the host machine.

Gitlab-runner

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 /srv/gitlab-runner/config/config.toml file.

```
[[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. Furthermore we map the Unix Socket to the Socket inside the container in the volume section. This is the reason why the runner will be using the docker daemon of the host machine. 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 it will be put on the default docker network.
The runners will later play a key role in the CI pipeline and make sure that the tasks are worked off.

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 two files:
1.) Turn on the Gitlab registry in the docker-compose file by modifying the Gitlab omnibus variable. This is the same as if you would change the gitlab.rb file. Furthermore you need to map the real physical ports to the ports in the Gitlab container.
2.) Add the registry as insecure-registry in /etc/docker/daemon.json file. This is because docker is expecting https instead of http by default.
Later in the pipeline, we will build the image, push it to the registry and deploy it on the machines in the cluster, all from within the pipeline.

Docker Swarm - Cluster

In our final system setup, we want to deploy an application onto a cluster of robots/machines. Therefore we needed a cluster orchestration tool. We had the choice between Kubernetes and Docker Swarm. Because there

already exists the service dind and our whole system is based on docker anyways, we used docker swarm to manage our cluster of machines. With Kubernetes there had also been some issues which will be explored later. Because one of our computers had a development version of docker engine installed, it was always breaking the daemon of other machines when trying to deploy a service. At the same time this computer was hosting the Gitlab and Gitlab-runner instance and therefore needed to have access to the cluster of machines. This meant that the computer needed to be at least manager status. Our solution was to create the cluster on a different machine and promote the computer with the Gitlab instance to manager.
We created a service with the image from the Gitlab registry on the other machine and afterwards we could update it from within the pipeline itself.

CI Pipeline

The pipeline is the place, where everything comes together and Continuous Deployment 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 Continuous Deployment:

```
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 url on which the Gitlab registry is sitting. 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 (called 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.

Final System Setup and 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. Then 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 and 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.

Monitoring

The idea for our monitoring system was to use Prometheus. This is a tool that collects data from a specified target, in our case from docker. Because we were out of time, we used a very sophisticated solution called swarmprom. In this solution the whole monitoring system was already build so 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.
We set the Gitlab instance to port 7070 and kept Prometheus running on port 9090. 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 it with Prometheuses own query language.
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 giving grafana access to Prometheus and entering the query string.

Practical use cases

- The final system setup is a first workflow setup 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 Deployment. 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.
- Because the application is run in containers, it will run in a lightweight fashion. This makes it possible that the application will not become too heavy.
- 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 in 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 software on the host machine and to connect to the external displays.
- If robot was temporarily down or container was shutting down, docker swarm will make sure, that the service is always deployed on every machine in the cluster.

Difficulties

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 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 our Gitlab instance 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

Kubernetes vs Docker Swarm

In our current version of our system we are using Docker Swarm instead of Kubernetes. But Kubernetes might be a more sophisticated cluster orchestration tool for professional usage. 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.

No development version

One factor that made our system much more unstable was the fact that the computer on which the Gitlab instance was set up used a development version of the docker engine. This computer was meant to be the swarm manager. 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 no development versions. Then one machine could be the host of the Gitlab and Gitlab-runner instance as well as the leader of the docker swarm. This would keep the whole logic of our system in one place.

Swarmprom vs GUI

In our final system setup we were using an already existing monitoring solution. Instead of using a version of another person, we could create our own monitoring system from scratch. Because of the time constraints this was not possible. One problem with swarmprom was that it was recording all but one of the computers of the cluster.

Use real robotic software

Up to now we only changed text files and deployed these changes on 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.

Ubuntu vs Windows

Another thing that slowed us down as a team was the fact that some computers did not have Ubuntu installed. It would have been best if every computer had ubuntu installed from the beginning.

Router scope

Up till now we are dependent on the router. This means, we can only operate if we are on the same subnet. Maybe some global url would be a solution for this problem.
Furthermore our mapping in /etc/hosts is bound to the ip which was given by the router. Therefore, if the lease is renewed and the router gives a different ip, we need to remap ip and name again.

Summary

In the end, we build a system for Continuous Deployment of an application over a cluster of machines. By using real robotic software, the system becomes a solution for Continuous Deployment of robotic software. Furthermore, we used an already existing monitoring solution to record the status of the machines in the cluster. At the same time, our system is rather unstable because of above mentioned reasons. This work can be a first step on designing a Continuous Deployment system for professional use cases.