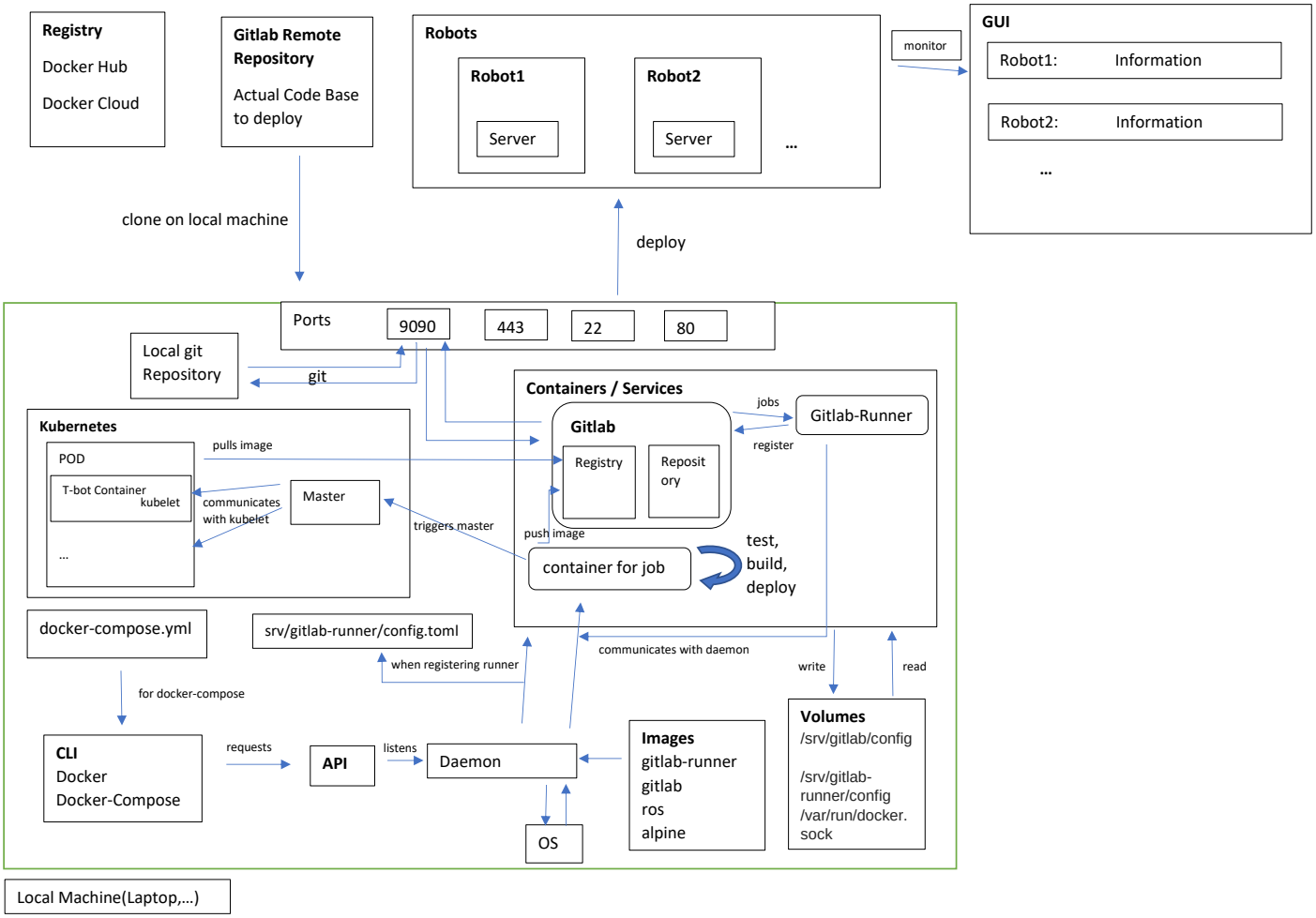

Project Outline

The objective of the whole project is to automatically test, build and deploy a code base located on a remote gitlab server onto a cluster of robots. We are currently at the point of deploying our application onto a cluster of machines.

System Setup

This section will summarize what we have done so far and the technical aspects behind it.
The starting point might be a code base containing the code that should be deployed. Instead of using an external git repository we initialized a local repository. Git will recognize any changes to the repository and can push these changes to an external repository if specified. At the same time, docker daemon spins up several containers. These containers are further specified in the docker-compose.yml file. In this file we can define several services/containers, the corresponding images and any other configuration. To make data persistent, docker uses a concept called volumes to store data.
One of the created containers is our local gitlab container, which sits on a specified port(e.g. port 9090). After initial registration, the gitlab instance will recognize our runner container as a runner that can pick up specified tasks. In our case the runner is configured in such a manner, that it is able to communicate with the /var/run/docker.sock file, which is basically the Unix Socket the docker daemon listens to by default. This makes it possible for the runner to communicate with the daemon within the container itself. Therefore the

runner can instruct the daemon to set up docker containers outside of the gitlab-runner container.
To trigger the CI/CD pipeline of gitlab, we need to define a .gitlab-ci.yml file and push it onto the master branch of our project. Gitlab will use this file to define several tasks which are initially set to a pending state and are supposed to be picked up by a runner. Our gitlab-runner instance will start working on these jobs by creating one container for every job it has to work on. This container provides an isolated clearly defined environment for our code to be tested, build and deployed on. The image it uses is either defined for each and every job in .gitlab-ci.yml file itself or in the config.toml file as a default image. The latter case is due to the registration process of our runner, where we set docker as executor and defined the default image(e.g. alpine,ros,...).
The jobs themselves are defined in the .gitlab-ci.yml file and can be assigned to the different stages of our CI/CD workflow. The runner will execute each job and stage independently. In our case these stages are the building, the testing and the deployment stage.

Connecting to Gitlab

Once the gitlab container is registered, the team can join the gitlab instance in their browser. Afterwards the admin of the gitlab instance can assign the people to the projects and give them a specific role. They will then be able to share the same repository and push their code to this repository.

Deployment Stage

The current strategy is to coordinate the robots inside a Docker swarm cluster. For simulation there will be some physical machines as workers connecting to the master node with docker. Once the workers are part of the cluster, they will even be recognized if master was temporarily unavailable. They will connect automatically with master due to stored credentials.
Our application will be deployed as a docker container and can be distributed over the cluster with a simple kubectl command. The images for these docker containers will be stored inside the gitlab registry.
These images must be built during the CI pipeline with our runners. The content of the image is defined in a Dockerfile, which specifies the application and must be available on the gitlab repository.
When pushing a .gitlab-ci.yml file to gitlab, this event will trigger a CI pipeline, where a runner will work on the different stages and jobs. After the testing stage, our gitlab-runner instance builds an image based on a specific application. This gitlab-runner will use a service called docker in docker (dind), so that it can actually build an image. This image will be uploaded onto the gitlab image registry. From there you can pull the contained images by means of some url. By entering the deployment stage, the gitlab-runner triggers the master to deploy containers based on the images in the registry over the cluster. This cluster will be tightly integrated with gitlab itself. The communication between the pipeline and the cluster can be defined with gitlab variables that can be used in the .gitlab-ci.yml file.
Because containers need to be built for our application to work, there will be docker installed on each and every worker. Docker containers, if set to privileged, can communicate with the underlying hardware. Therefore it is possible to simulate applications for robots within docker containers in a lightweight predefined environment.
Because the project is heading towards the end, we have to implement a functioning system. Because there are some issues with the deployment over Kubernetes, this option will be delayed. For Kubernetes initialize a cluster with kubeadm and join over kubectl. You can even make the cluster recoverable.

Advantages and Disadvantages

Advantages:
1.) There will not be any ssh authentication because the robots will be organised within the Kubernetes cluster. Therefore the maintenance of the connections will be reduced. Everything will be coordinated with one (few) master node(s).
2.) If it is possible to trigger the master of the cluster within the .gitlab-ci.yml file we can basically achieve Continuous Deployment.
3.) With kubeadm it is very easy to set up a master node and connect different nodes to the cluster. If the cluster is configured to be recoverable, workers connect themselves to the cluster once joined manually.
4.) When using the gitlab registry as storage type for our application, the master node of our cluster can tell the robots to pull this image.
5.) The final system can run on local machines without being dependent on different physical machines or external entities like DockerHub. The application storage, namely the gitlab registry as well as the whole codebase as well as the master of the Kubernetes cluster can be organised on a single machine.
Disadvantages:
1.) For simulating our system it is best to use minikube to run Kubernetes locally. Minikube is a single-node Kubernetes cluster inside a VM. Because there is only one single node, we must organize the robot containers in a pod. This restriction is a problem for the final system, where multiple physical machines run as multiple nodes in the cluster.
2.) Minikube is running inside a VM. We initially chose docker because we wanted to go without the disadvantages of VM and hypervisor. After the simulation of our system locally we will have to swap minikube with a real Kubernetes option.
3.) Currently we are using kubeadm for setting up our cluster. Unfortunately kubeadm demands to turn off swap for the whole system. This may or may not slow down the master node drastically. If we consider that the master also contains the whole system, it is definitely a point to keep in mind.
4.) When using real physical machines, they need to have docker installed for building the images.

Summary

For our final system we have five major components that must be connected with each other and work together properly:
1.) The first component is Gitlab, which is run inside a Docker container and defines the center of gravity for our project. The gitlab container will store all of our code and is connected to all of the other components.
2.) The next essential component is the Gitlab runner container. This container makes it possible to run testing environments for our code inside a Docker container within the host machine itself. These runners are the workers that bring the CI/CD pipeline into being.
3.) The pipeline itself is defined inside the .gitlab-ci.yml file. Every time there is an update to the repository a pipeline gets triggered and runners work off the .gitlab-ci.yml file.
4.) The gitlab registry can be integrated into our gitlab container and will serve as central image registry for our project. For deployment we can therefore distribute containers based on images inside this image registry. This makes it possible for us to create a clearly defined environment for our application independent of the underlying hardware. Furthermore we have the whole functionality of our applicaiton concentrated inside one container.
5.) The last big component is a cluster deploying the applications over docker Containers. Our current version is run by docker swarm, but might be replace by a Kubernetes cluster in the future. The Kubernetes cluster can be tightly integrated with our gitlab container. Therefore it is possible to trigger the deployment inside the .gitlab-ci.yml file. But with service docker in docker, docker swarm is much easier to use, because we can

interact with the docker daemon within the .gitlab-ci.yml file.
For now the master of the cluster will contain the whole system.

Possible future projects

- construction of a GUI for monitoring states of different robots in cluster