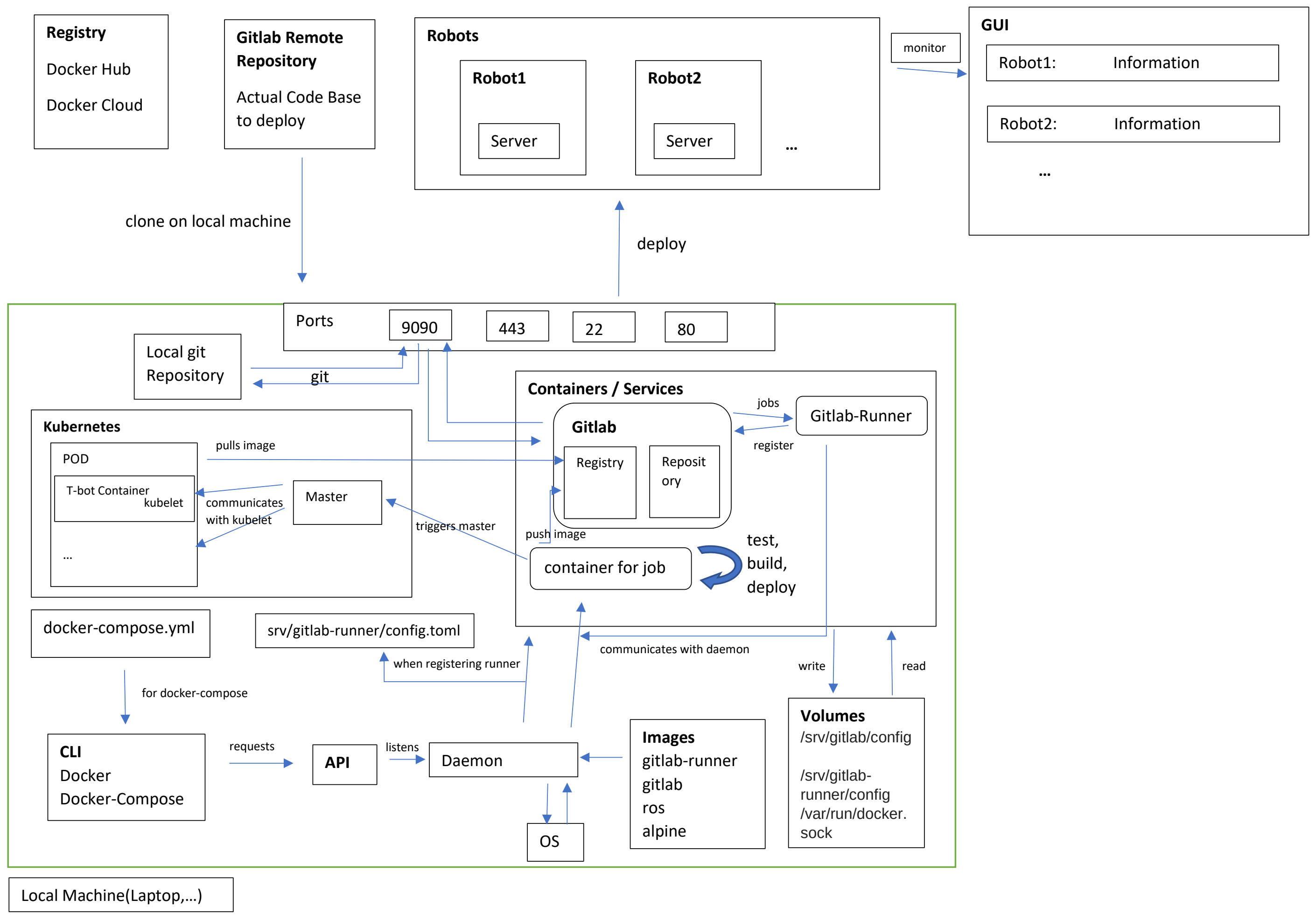

Project Outline

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. In the last few weeks we have set up our system on our local machines to the point of deployment. The missing parts are the usage of the actual code base for the robots, the deployment onto the robots and the monitoring aspect containing the corresponding GUI.
In this elaboration we especially should research some possible solutions for the deployment stage of our project.

Local System Setup

At this point in the process we kept the scope of the project on our local machines. 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). Ports are especially important when trying to deploy our code over ssh. 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. In our case these stages are the building, the testing and the deployment stage.
The runner will execute each job and stage independently. In the build stage, the source code will be compiled and generated to execute. After building the application it needs to be tested. For this to happen, there must be many files which penetrate the source code with test cases and compare the results. At last there is the deployment stage.

Deployment Stage

The idea is that within the deployment stage the runner will distribute the tested and compiled code onto several robots. We had to research on possible deployment techniques which could be of use to meet the requirements of the project.
My first approach went something like: Use ssh to connect to remote servers, afterwards deploy/copy files with some bash command. To establish the connection, there are basically two options, either by entering a password or by using ssh key authentication. The best practice would probably be to create an ssh key pair and copy the public key onto the remote server. Afterwards the connection can be established with a simple bash command, which can be put into the .gitlab-ci.yml file. After the connection is provided we can use scp, rsync or sftp to deploy/copy the compiled files onto our robots/remote servers. Unfortunately there were some issues by simulating this strategy with docker containers and ssh. It still might be a viable solution for our project, so it can be pursued later.
Another strategy is to coordinate the robots inside a Kubernetes cluster. For simulation there will be some containers representing turtlebots and one additional master node, which communicates with the "robots". This communication takes place over some service called kubelet which sits on every robot. The containers will be organised in a pod, so that you can define an environment for them once. 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 an image that contains the service 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 command containers in the cluster to pull a specific image/our application from the registry. Because containers need to be built for our application to work, there will be docker installed on each and every container robot. 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. When using real robots, we might want to switch to multiple node clusters, because pods rather run containers instead of real physical machines.

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). This master node will be the interface of our local machines to each and every robot inside of the cluster.
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.) It is fairly simple to add a new robot/node to our cluster.
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 without further ado.
5.) The simulation of the robots with containers is pretty equivalent to the deployment onto real machines. Containers can not just be swapped with real machines but the simulated structure might be expanded, so that we have a cluster with multiple nodes. Maybe it is even possible to organize the containers not within a pod, but each and every container as a single node. This would approach our simulated system to the final system.
6.) 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.) Currently the robots must build the containers themselves. In our simulation we can use the docker socket of the installed VM(e.g. virtualbox) used by minikube, a local Kubernetes application. Then docker will not have to be installed on the robot containers. But when using real physical robots, they need to have docker installed for building the pulled images. Maybe the effort is minimized by coordinating the docker installation with master but it still increases the complexity of our system.
2.) 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.
3.) Minikube is running with 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.

Final System Setup

For our final version, we now have to include the actual code base from the remote gitlab repository that should be deployed. Therefore we have to clone the remote repository onto our local machine, so that we have a local git repository. Afterwards we push this new repository onto our local gitlab instance inside the container. As soon as .gitlab-ci.yml file is recognized by gitlab, the runner container should process the tasks defined in this file.
Another option would be to connect the runner to the remote gitlab repository. But then we would not need the gitlab container instance.

Monitoring Stage

This phase contains the construction of a GUI, which displays status information about the different robots. Because there is no real specification how this should be done, the elaboration will not explore this stage any further.